UNIVERSIDAD CARLOS III DE MADRID
Escuela Politécnica Superior
Ingeniería en Informática

Proyecto de Fin de Carrera

Implementación de una arquitectura para el desarrollo de comportamientos para StarCraft apoyado en PELEA

Alumno: Javier Márquez Colás
Directores: Daniel Borrajo Millán y Moisés Martínez Muñoz

Agradecimientos

A mis padres, por ser capaces de no presionarme y apoyarme durante el viaje, aun cuando los tropiezos fueron evidentes.

A mi hermana por tener la paciencia para aguantarme en esos días en el que el estrés y el agobio hacían presencia.

A mi tutor, Daniel, por tener la paciencia para guiarme y ofrecerme un proyecto tan interesante como divertido y educativo. Gracias también a Vidal y Moisés por sus explicaciones y ayuda con PELEA.

A los compañeros con quienes he podido compartir grandes momentos, en especial a Luis, gran compañero y mejor persona con quien en tantas prácticas he tenido el placer de haber trabajado.

Mención especial también para los compañeros y amigos que al final de la carrera trabajamos juntos y que me han obsequiado con recuerdos memorables, Verónica y Alejandro.

Tampoco puedo olvidar agradecer Cris y Fran los grandes momentos que pasamos ya fuera de la universidad.

A María, por ser la persona más optimista y luchadora que he conocido capaz de ser además generosa y de comprenderme.

Por último, a Roberto, por tener la sabiduría y paciencia para hacerme ver con sus consejos.

Resumen

Los juegos de estrategia en tiempo real suponen un cambio en el concepto de jugabilidad al ofrecer a los participantes la posibilidad de realizar sus movimientos al mismo tiempo. Además, se distinguen de los juegos de estrategia en turnos cómo puede ser el ajedrez, en que el número de unidades que cada jugador puede controlar varían conforme el tiempo avanza tanto en número como en variedad de tipos según el jugador decida progresar en la partida. Estas características componen un entorno de trabajo muy complejo a la hora de desplegar jugadores que utilizan inteligencia artificial para tomar decisiones. Estos jugadores artificiales deben ser capaces de adaptarse a las variaciones en el entorno así como a las estrategias empleadas por los jugadores humanos y el resto de jugadores artificiales. Sin embargo, este aspecto ha sido relevado a favor de otros tales como el modelado 3D y los gráficos de los juegos. Esto ha llevado a que cada vez más jugadores demanden una IA más competitiva que sea capaz de suponer una dificultad cuando ya se han jugado varias partidas.

Una forma de mejorar los comportamientos de los jugadores automáticos, es mediante la utilización de la Inteligencia Artificial. En este documento se presenta el desarrollo de un sistema de control basado en Planificación automática para el juego StarCraft. La planificación es un área de la inteligencia artificial que permite la generación de planes de acciones para resolver un problema dada una representación del entorno. Con el fin de desplegar la Planificación Automática como modelo de razonamiento para StarCraft se ha utilizado la arquitectura PELEA, para la cual se ha implementado un conector con el fin de interactuar con el juego capaz de transformar la información del juego en un estado PDDL necesario para su posterior transformación en XPPDL, el formato que PELEA utiliza internamente además de ser capaz de ejecutar las acciones que PELEA le indique en el juego. Para permitir esta interacción se ha utilizado BWAPI y ChaosLauncher. BWAPI es un API para la interacción con StarCraft y ChaosLauncher es el programa que permite la interacción.

Abstract

Real time strategy games suppose a change in the concept of gameplay by offering the players the possibility to perform their moves at the same time. In addition, they differ from the turn based strategy games like chess in that the numbers of units each player can control vary as time goes on in quantity and types as the player makes decisions towards the match. These characteristics create a complex framework when deploying players who use Artificial intelligence to make decisions. These artificial players must adapt to changes in the environment and the strategies the human players and the rest of the artificial players. However, this aspect has been replaced for others like 3D modeling and game graphics. This produced a growing number of players asking for more competitive AI able to suppose a challenge after several matches have been played.

One option to improve the behavior of automated players is by using Artificial Intelligence. This document presents the development of a control system based on Automated Planning for the StarCraft game. Planning is a field of Artificial Intelligence which allows the generation of action plans to solve a problem given a representation of the environment. With the goal of deploying Automated Planning as the reasoning model for StarCraft the PELEA architecture has been used and an adaptor has been implemented. This adaptor is able to interact with the game and capable of transforming the information of the game into a PDDL state required for its later translation to XPDDL, the format PELEA used internally in addition to being able to execute into the game the actions PELEA orders. To allow this interaction BWAPI and ChaosLauncher have been used. BWAPI is an API for the interaction with StarCraft and ChaosLauncher is the program who allows the interaction.

Contenido

Índice de figuras

Índice de tablas

Capítulo 1: Introducción

Los videojuegos son un valor en alza en la industria del entretenimiento que disponen de grandes presupuestos para su desarrollo (1). Dentro de ellos se encuentran los populares juegos en tiempo real o ***R**eal **T**ime **S**trategy* (RTS). Estos juegos se distinguen de los juegos de estrategia por turnos por que no se dispone de información sobre en qué periodo de tiempo se pueden realizar las acciones o los movimientos ya que estos son ejecutados en el momento en que son ordenados de forma simultánea. Además, los jugadores se han de enfrentar entre ellos en un escenario o mapa ejecutando una gran cantidad de acciones por minuto mientras combinan estrategias a largo y corto plazo que les permitan obtener una ventaja frente a su adversario tanto en un enfrentamiento puntual como en los futuros. Para ello deben de gestionar la extracción de recursos, la creación de unidades y edificios y el desarrollo tecnológico que les permita mejorar o crear nuevos tipos de unidades (2) a la vez que ordenan a sus unidades los movimientos que han de ejecutar. Estas características incrementan la complejidad del juego a la hora de desarrollar sistema de juego artificial. Estos jugadores han sido tradicionalmente abandonados a favor de otros aspectos del juego tales cómo los gráficos o el modelado 3D. No obstante, los jugadores de estos videojuegos demandan cada vez jugadores artificiales más complejos que sean más difíciles de derrotar y que por tanto conduzcan a una experiencia de juego más completa. Tradicionalmente estos jugadores están basados en programación estática a través de *scripts*, reglas o máquinas de estados finitas para la toma de decisiones (3). Este tipo de aproximaciones conduce a que al cabo de unas partidas, un oponente humano sea capaz de reconocer y predecir el comportamiento del jugador artificial resultando en una victoria sencilla debido a que la capacidad para adaptarse a los cambios del entorno de estos sistemas es limitada.

Entre estos juegos se encuentra StarCraft (4), un juego de reconocido éxito mundial por su competitividad y complejidad que integra las mecánicas clásicas de los juegos de su género además de introducir nuevos componentes. Entre los retos que StarCraft proporciona se encuentran el manejo de una cantidad elevada de unidades en mapas extensos capaces de soportar el juego simultáneo de varios jugadores. StarCraft presenta tres aproximaciones diferentes a la forma de juego representada en las tres diferentes razas que un jugador puede seleccionar al jugar la partida y que se traducen en diferentes unidades, habilidades para cada una de ellas y formas de construcción. Estas diferencias proporcionan un elevado número de estrategias de desarrollo que dependerán de la raza que se esté jugando, la del oponente u oponentes, y otras consideraciones tales como la forma del terreno del mapa, la exploración del mismo y sus recursos disponibles.

Para poder solventar los problemas que las características de StarCraft plantean y generar comportamientos que se asemejen a los de un jugador humano, se propone la utilización de la Inteligencia Artificial. La Inteligencia Artificial (IA) es un área de la informática que estudia y diseña agentes inteligentes. El concepto de Inteligencia Artificial fue introducido por Alan Turing en 1950 cuando formuló la pregunta “¿puede pensar una máquina?” (5). Inicialmente esta se orientó a la resolución automática de demostraciones matemáticas o a la creación de jugadores para juegos clásicos como el ajedrez o las damas. Su evolución ha derivado en la aparición de nuevas áreas como el aprendizaje automático, la robótica inteligente o la planificación automática (6). La planificación automática es una área de la inteligencia artificial que consiste en la creación de un plan de acciones que ejecutados de forma secuencial son capaces de alcanzar unos objetivos a partir de un estado inicial del entorno (7). Esta técnica puede ser utilizada para la generación de un sistema de razonamiento de forma que los planes generados induzcan la sensación de que el jugador automático tiene un objetivo propio y es capaz de adaptarse al entorno. Así, se desligan del concepto limitado de jugador automático como un ente capaz de realizar una tarea simple, ya que pueden ofrecer soluciones complejas.

Una arquitectura capaz de realizar la planificación, de ejecutar los planes obtenidos y de controlar la ejecución de los mismos es PELEA (8). PELEA es un acrónimo de sus siglas en inglés "***P**lanning, **E**xecution and **LE**arning **A**rchitecture"* que ha sido desarrollado entre los integrantes del grupo ***P**lanning and **L**earning **G**roup* (PLG) pertenecientes a laUniversidad Carlos III de Madrid e investigadores de las universidades Politécnica de Valencia y de Granada.

1.1. Objetivos

El proyecto tiene como objetivo principal el desarrollo de un sistema de control para un jugador autónomo (bot) para el juego StarCraft mediante el uso de planificación automática. A continuación se presentan de forma detallada los distintos objetivos en los cuales se divide el objetivo principal de este proyecto.

1. Estudio y análisis de StraCraft: para el desarrollo del sistema de control es necesario analizar el funcionamiento del juego sobre el cual será desplegado. Así como las diferentes formas que existen para la generación de jugadores automáticos, el lenguaje de programación para su desarrollo, los programas necesarios para su funcionamiento y la plataforma para la implementación.

2. Estudio y análisis de la planificación automática: para el desarrollo de un sistema de control basado en Planificación automática, es necesario analizar el funcionamiento de las técnicas más apropiadas, las herramientas que existen actualmente y si existe algún tipo de modelos de control que puedan ser utilizados en este proyecto.

3. Diseño e implementación del sistema de control:

    3.1. Definición del dominio de planificación para el juego StarCraft: es necesario definir el dominio sobre el cuál se va operar en lenguaje PDDL (***P**lanning **D**omain **D**efinition **L**anguage*), definiendo las acciones, funciones y hechos que componen la descripción del juego sobre el cuál se va a operar. Además, PELEA requiere de la redefinición de las acciones implementadas que el *bot* puede llevar a cabo en este dominio para dotar de una mayor complejidad al mismo, pudiéndose definir de esta manera acciones que tengan como objetivo el control interno del sistema o el estado además de la actuación del propio *bot* del juego.

    3.2. Creación de un sistema de traducción: un sistema que permita transformar el estado del juego en una representación que pueda ser utilizada por un planificador. En este punto cabe destacar la necesidad de la traducción del estado del juego a una representación intermedia en basada en los eventos que se produzcan en el juego y que estén contemplados en el dominio. Esta representación intermedia es importante para cumplir con el objetivo de minimizar los retardos al obtener el estado que puede ser consultado en cualquier momento mediante una petición de PELEA.

    3.3. Creación de un conjunto de acciones que permitan crear planes para el juego, así como un sistema de traducción automática de dichas acciones: será necesario definir los actuadores capaces de llevar a cabo las acciones definidas en el dominio mediante implementación de las mismas dentro del sistema y posteriormente, será necesario definir la traducción de las acciones recibidas en los actuadores concretos que controlarán el juego, así como las post condiciones derivadas de la ejecución de las mismas.

4. Integración del sistema de control con la arquitectura PELEA: será necesario ejecutar un plan recibido, así como controlar que las acciones se han podido llevar a cabo o se han cumplido las post condiciones en caso de ser necesario.

5. Desarrollo de un conjunto de experimentos para comprobar el correcto funcionamiento del sistema de control desarrollado

   5.1. Diseño de un conjunto de experimentos: para una correcta comprobación del funcionamiento del sistema desarrollado es necesario crear un conjunto de experimentos que comprueben la funcionalidad del mismo.

   5.2. Análisis de los resultados: con los resultados obtenidos de la experimentación con las pruebas diseñadas se puede obtener conclusiones sobre la aplicación del paradigma utilizado y sobre el funcionamiento y rendimiento del sistema.

6. Desarrollo de la documentación: tras el desarrollo del proyecto es necesario generar la documentación que recoge la información tanto de su desarrollo como de su funcionamiento e implementación.

1.2. Estructura del Documento

Este documento se divide en siete capítulos, de los cuales el último alberga los distintos anexos del proyecto. A continuación se realiza una descripción del contenido de cada uno de ellos:

- En el primer capítulo se presenta una breve introducción al proyecto desde la perspectiva actual de los juegos en tiempo real. A continuación, se describen los objetivos a cumplir del proyecto.

- El segundo capítulo se corresponde con el marco teórico del proyecto. Inicialmente se describen de forma detallada los juegos de estrategia en tiempo real, los problemas que surgen cuando se aborda la creación de un jugador automático y las soluciones que se han utilizado. A continuación se presenta una descripción del juego de estrategia objetivo de este proyecto, StarCraft, destacando los elementos de su interfaz, los recursos, las unidades y los mensajes y parte de las mecánicas diferentes que rigen su comportamiento. Para finalizar se realiza una descripción de la planificación automática realizando una descripción del modelo conceptual, el lenguaje de representación utilizado y los algoritmos utilizados así como la descripción detallada de la arquitectura PELEA utilizada para la planificación y ejecución de los planes resultantes.

- En el tercer capítulo del documento se describe la estructura de la arquitectura de control desarrollada para el *bot*, describiéndose de forma detallada todos los elementos que la componen.

- El cuarto capítulo se corresponde con la evaluación del sistema, realizándose una descripción detallada de las pruebas diseñadas y de los resultados obtenidos tras su realización.

- El quinto capítulo se corresponde con la información sobre la gestión del proyecto y sus costes.

- En el sexto capítulo se presentan las conclusiones obtenidas tras la realización del proyecto, los problemas surgidos durante el desarrollo y las posibles líneas futuras de trabajo.

- Por último se incluye un conjunto de anexos en el séptimo capítulo. En ellos se incluye una descripción del proceso de instalación del sistema tanto para uso como para desarrollo, un manual de usuario y una descripción detallada de BWAPI, el API utilizada para comunicarse con StarCraft. Además, se incluye una descripción detallada del programa utilizado para la inyección del código generado en el juego, ChaosLauncher y del juego StarCraft. Finalmente, se presenta el diseño detallado del sistema.

Capítulo 2: Estado del arte

Los juegos por ordenador se corresponden a una industria de entretenimiento en crecimiento y constante evolución que generan nuevos retos para sus desarrolladores no sólo en el apartado gráfico, sino también en el desarrollo de jugadores automáticos capaces de jugar a los mismos de forma competitiva. Esta evolución ha propiciado la creación de nuevos géneros en los juegos en función de la forma de juego.

Estos géneros varían en función de la interacción del jugador y del planteamiento del juego. Entre estos géneros se pueden encontrar juegos de estrategia, conducción, lucha, simulación…. Además, pueden clasificarse en función de si la interacción con el jugador se produce por turnos o en tiempo real.

2.1. Juegos de Estrategia en Tiempo Real

Los juegos de estrategia en tiempo real suponen un cambio de paradigma sobre los juegos basados en turnos, ya que al tratarse de dominios continuos, la acción tiene lugar en tiempo real, sin disponerse de un lapso de tiempo por turno en el que poder desarrollar una estrategia, analizar la estrategia del contrincante o definir un orden de actuación. En estos juegos las decisiones han de ser tomadas mientras se juega, al mismo tiempo que se ejecutan las acciones y a un ritmo rápido.

Son juegos que presentan la dificultad añadida de tener que controlar simultáneamente múltiples edificios y unidades a la vez que se distribuyen los distintos recursos que se extraen por las unidades entre la creación de nuevas unidades, edificios o el desarrollo tecnológico y de habilidades en las unidades del jugador. Todo ello coordinado con o sin aliados y contra uno o varios contrincantes sobre un determinado mapa o escenario.

Estos mapas están compuestos de casillas por las que las unidades pueden moverse y los edificios ser construidos si el tipo de terreno lo permite. Además, ciertos juegos ofrecen la posibilidad de definir características que influyen directamente en el comportamiento de las unidades que estén traspasándolas. Los comportamientos más habituales son:

- Impedir el movimiento: bien porque se trate de terreno infranqueable como puede ser una muralla o bien porque para traspasar esas casillas se requiera de otras unidades capaces de transportar en su interior a las que no son capaces de moverse por ese terreno. Un ejemplo de lo anterior es el agua, las unidades terrestres no pueden atravesarla salvo que se hallen en el interior de un barco con capacidad de transporte de unidades terrestres.

- Ralentizar el avance: ciertos terrenos como puede ser una ciénaga o terreno elevado, suelen impedir el movimiento de las unidades en una determinada cantidad, haciendo que atravesarlas sea más costoso.

- Otras ventajas/desventajas: un terreno elevado suele favorecer a las unidades a distancia que se encuentren en ellas a la vez que genera una desventaja en las mismas que se encuentren en el terreno inmediatamente inferior.

Finalmente, el jugador suele disponer de un mini-mapa con el escenario que indica eventos que se producen, tales como ataques o unidades recién creadas. Este mini-mapa muestra también las unidades propias y enemigas si estas últimas se encuentran visibles. La visibilidad viene determinada en tres categorías:

- Terreno inexplorado: se trata de terreno que no ha sido explorado por ninguna unidad, y que aparece como una zona negra en el mini-mapa.

- Terreno explorado: se va descubriendo conforme las unidades exploran el terreno y se queda con "niebla de guerra" cuando no se encuentra en el radio de visión de ninguna unidad.

- Terreno visible: este terreno se encuentra en el área de visibilidad de alguna unidad y se dispone de toda la información posible.

Estas características san hecho de los juegos en tiempo real un dominio difícil de dominar por los humanos (9) y una herramienta para la investigación en Inteligencia Artificial muy importante ya que la capacidad de juego a nivel experto en este tipo de juegos requiere de la ejecución de cientos de acciones por minuto mientras que se trabaja por lograr unos objetivos globales. El control para la realización de dichas acciones se ha dividido en:

- Micro control: hace referencia al control individual de las unidades. Dentro del micro control, se suelen incluir los siguientes campos:

  - Control individual de la unidad: en este nivel se controla cada unidad de forma individual con el objetivo de mejorar su rendimiento en combate o al moverse por el mapa.

  - Control táctico: el control táctico define la aproximación que se sigue al ordenar el ataque sobre una posición y durante el desarrollo de una batalla.

- Macro control: el control "macro" hace referencia al control estratégico de la partida, es decir, qué se va a crear, cuándo, dónde, con qué recursos, cuándo atacar, realizar expansiones en una base …

2.1.1. Retos de los juegos en tiempo real

Entre los retos que ofrecen los juegos en tiempo real se distinguen los siguientes:

- Tiempo real: se trata de juegos en que las acciones han de ser realizadas a un ritmo muy elevado, de forma continua y simultánea. Además, son entornos hostiles donde los enemigos pueden cambiar el estado del juego de forma asíncrona.

- Falta de certeza: en los RTS no se dispone de la información completa del mapa, ni del enemigo, por lo que las decisiones han de ser tomadas con información parcial e incompleta.

- Modelado y aprendizaje del oponente: en estos juegos, se ha de reconocer las debilidades en la estrategia del rival y explotarlas, siendo capaces por tanto de analizar la información observada del rival y de realizar predicciones basadas en las mismas.

- Control de los recursos: la gestión y el buen uso de los recursos es crucial para obtener la victoria. Decidir qué unidad ha de construirse en qué momento y lugar toma especial importancia cuando se tiene en cuenta el recurso del tiempo. Otros recursos que se han de gestionar de manera eficiente es la búsqueda de un camino si no viene integrada dentro del propio juego, y la gestión del espacio para el emplazamiento de los edificios.

- Colaboración: al tratarse de juegos en los que se puede contar con otros jugadores aliados, la colaboración entre ambas partes surge no solo de la coordinación de los ataques, si no en la combinación de las unidades de los distintos jugadores para conformar un ejército capaz de vencer a los rivales. Además, tareas como la exploración pueden desarrollarse de forma coordinada no siendo necesario que cada jugador explore por completo su mapa.

- Complejidad del mundo: los mundos constan de un gran número de casillas en las rejillas que los conforman. Además, durante una partida se pueden contar con cientos de unidades bajo el control de cada jugador, por lo que el espacio del problema posee un tamaño considerable.

2.1.2. Técnicas de la IA aplicadas

Históricamente, los diseñadores de juegos han creado la ilusión de inteligencia a través de *scripts*. Un *script* es una lista de órdenes que se ejecutan típicamente desde un archivo. Sin embargo, aplicado a juegos, un *script* es el conjunto de reglas que rigen el comportamiento de una unidad. De este modo, se integra con el sistema de juego y sus mecánicas, que son también reglas, para generar un comportamiento que se ha denominado la "inteligencia artificial" del juego, o el jugador controlado por el propio juego.

El problema del *scripting* es que para ser efectivo, debe ser complejo, pero la complejidad da la oportunidad para la debilidad y la predictibilidad al final (10). En un momento, el jugador humano será capaz de reconocer y explotar esas debilidades, eliminando la ilusión de un contrincante humano. Para solucionar este problema, se han utilizado técnicas como el *scripting* dinámico que consiste en la generación de nuevos *scripts* basados en reglas y en función del éxito de aplicaciones anteriores de las mismas. De esta forma, los *scripts* son capaces de adaptarse rápidamente y de desarrollar comportamientos y tácticas complejas que son ofrecen contra-estrategias al oponente. Sin embargo, se han aplicado otras técnicas de la Inteligencia Artificial para superar los retos que este tipo de juegos ofrecen tales como:

- Computación biológica y optimización estocástica: la computación biológica ha sido usada ampliamente en los juegos RTS y ha sido combinada con técnicas de optimización estocástica para mejorar su rendimiento. Se ha usado principalmente para aprendizaje.

- Razonamiento basado en casos y aprendizaje por refuerzo: han sido utilizados para aprendizaje pero principalmente para la selección de los planes a ejecutar.

- Simulación: el algoritmo de Monte-Carlo ha sido utilizado para realizar planificaciones rápidas y simulaciones sobre los posibles desarrollos del juego.

2.2. StarCraft

StarCraft es un juego de estrategia militar en tiempo real desarrollado por la compañía Blizzard Entertaiment® en 1997. Posee un paquete de expansión, Broodwar, que añade nuevas unidades y parches que equilibran aún más las diferentes razas. Es un juego reconocido mundialmente, en el que se realizan campeonatos. Es especialmente popular en Corea del Sur, donde dichos campeonatos son retransmitidos por la televisión. Además, en ellos se suelen enfrentar jugadores en partidas de uno contra uno, aunque algunos torneos ofrecen la posibilidad del juego en equipo.

StarCraft está ambientado en una galaxia lejana, conocida como el sector "*Koprulu*" lugar donde las tres razas que componen los diferentes estilos de juego compiten por sobrevivir. Dada la complejidad del mismo y los objetivos del presente PFC se eligió la raza Terran como destinataria de los *scripts* de ejemplo del *bot*. A continuación se procede a mencionar los aspectos relevantes de la mecánica del juego y de la raza Terran que aplican en este proyecto.

2.2.1. Interfaz del juego

Tras iniciar una partida en un mapa cualquiera, se suele obtener una situación inicial cómo la mostrada en la figura 1.

Figura 1: Interfaz del juego

A continuación se va a analizar cada uno de los elementos que se han tenido en cuenta para este *bot* y se explicarán el resto de elementos de la interfaz: Recursos, unidades, y mensajes.

2.2.1.1. *Recursos:*

Los recursos extraíbles en StarCraft se muestran en la figura 2 y destacan por su limitada cantidad y su pequeño número. Estos recursos son utilizados para la construcción de edificios, entrenamiento de unidades e incluso para algunas habilidades. Se distinguen los siguientes:

Figura 2: Recursos de StarCraft

Donde:

1. Depósitos de mineral: los depósitos de minerales son los lugares donde los trabajadores, o SCV para la raza Terran, pueden extraer este recurso situándose en una posición contigua a la del depósito y recolectarlos en un tiempo. Cuando una veta de mineral ha sido explotada completamente, desaparece dejando el terreno libre.

2. Géiser de vespeno: en este emplazamiento se puede construir un extractor de vespeno, capaz de sintetizar el gas y permitir a un SCV recogerlo. Cuando un géiser se vacía, la producción se reduce en un cuarto de la producción normal.

3. Panel de recursos: en este panel se recogen las cantidades actuales de recursos que posee el jugador así como la población actual y la máxima que es capaz de mantener.

La extracción de recursos se realiza en grupo de 8 unidades de recurso. Además, este recurso deberá ser transportado por un trabajador al centro correspondiente para que dicha cantidad sea añadida a los recursos disponibles. Finalmente, solo un trabajador puede en un momento dado extraer recursos tanto de un depósito de mineral como de un extractor de vespeno, aunque si éste se encuentra de camino a entregar los recursos o el depósito, otra unidad puede recolectar del mismo depósito o extractor.

En el caso de la población, es necesaria la construcción de un edificio, el depósito y se tiene un límite de población ampliable hasta 200.

2.2.1.2. *Unidades*

Al comienzo de una partida se suele contar con las unidades que se muestran en la figura 3.

Figura 3: Unidades iniciales

Donde:

1. Centro de mando: es el centro para la raza Terran. Es un edificio capaz de almacenar recursos y construir trabajadores, entre otras cosas.

2. Terran SCV: son los trabajadores de la raza Terran, unidades capaces de construir, extraer recursos y reparar unidades mecánicas, incluidos otros SCV pese a ser estos

catalogados como orgánicos. La extracción de recursos una vez ordenada se realiza de manera automática por dicha unidad hasta que otro comando le sea ordenado.

3. Paneles de control de unidad: en ellos se puede encontrar la información referente a la unidad seleccionada. En el panel de la izquierda puede comprobarse su estado, sus puntos de resistencia restantes y totales así como la cantidad de armadura o ataque y el nivel de investigación en estos.

   En el panel de la derecha se encuentran las órdenes y habilidades especiales de la unidad representados con iconos. En este caso corresponden a un SCV, aunque también se representan en dicho panel los del resto de unidades, incluidos edificios. De arriba abajo y de izquierda derecha, mover a un punto, parar, atacar un punto, reparar, recolectar, construir edificios básicos y avanzados.

4. Retrato: retrato animado de la unidad.

Las unidades pueden desplazarse por el terreno siempre y cuando haya camino disponible, el terreno sea transitable o sea una rampa. Las unidades no pueden atravesar edificios ni ocupar una misma posición. Esto no aplica en las unidades aéreas.

2.2.1.3. *Mensajes*

Los mensajes que el juego proporciona se enmarcan en las siguientes zonas representadas en la figura 4.

Figura 4: Interfaz de StarCraft

Donde:

1. Mensajes: En éste área se recogen tanto los mensajes que el jugador envía, cómo los que recibe.

2. Mini mapa: En él se reciben notificaciones de ataque y creación de unidades así como servir para la consulta de la información del mapa del juego que haya sido descubierta vía explotación. El área cubierta de negro indica territorio inexplorado, los puntos verdes las unidades propias, los azules depósitos de mineral y cualquier otro color unidades enemigas visibles.

Es importante destacar que la información de las unidades ocultas por la niebla de guerra no se muestra en el mini mapa.

2.2.2. Mecánica del juego

Entre las peculiaridades del juego se puede destacar las siguientes:

- Las unidades tienen distinto coste en población.
- Existen unidades aéreas capaces de sobrevolar el terreno, u otras unidades.
- Cada raza además posee un transporte aéreo diferente.
- Ciertos ataques son más efectivos contra unas unidades que contra otras, basadas en el tipo del ataque y el tamaño de la unidad objetivo.
- Las unidades poseen habilidades que pueden ser desarrolladas mediante tecnología y que pueden requerir costes como energía, vida o la habilidad de moverse mientras está activada.
- Ciertas habilidades solo pueden ser utilizadas en unidades de un tipo, mecánicas u orgánicas.
- Existe la posibilidad para ciertas unidades de hacerse invisible o hacer invisible a otras, así como habilidades o unidades capaces de detectar otras invisibles.
- El juego tiene en cuenta el desnivel del terreno para las batallas a distancia y los objetos como árboles que puedan interponerse.
- Este terreno en su nivel interno trabaja como una rejilla de posiciones a tres niveles de tamaño.
- Existen habilidades que solo se pueden usar contra una raza, u edificio concreto.
- Las diferencias de juego entre cada raza se pueden notar hasta en la creación de unidades y construcción de edificios, siendo diferente para cada una.
- Existe niebla de guerra, o la pérdida de visión de lo que ocurre en un terreno descubierto cuando una unidad se aleja. Este rango de visión puede ser alterado por unidades en el juego.

- Las unidades son capaces de encontrar un camino hasta el punto indicado para el desplazamiento, si lo hubiere.

- Las unidades tienen una cola de órdenes que van ejecutando secuencialmente, aunque se puede reiniciar a una orden u otra cola de órdenes.

2.3. Planificación Automática

La planificación es un acto deliberativo que busca la resolución de un problema o estado inicial anticipando los resultados de las acciones que se pueden realizar. Es decir, busca alcanzar unos objetivos mediante la toma ordenada de decisiones. La planificación automática es un área de la inteligencia artificial que estudia este proceso de forma computacional. Dentro de la planificación se encuentra el tipo de planificación determinista que considera que el entorno en el cual se opera es completamente conocido y en el que la una acción produce igual resultado al ser ejecutada con los mismos parámetros. Esta aproximación es la que se utiliza en la planificación clásica. Otro tipo de planificación es la probabilística en la que el entorno es desconocido completamente y las acciones pueden producir diferentes transiciones de estado en función de una distribución probabilística. También se encuentra la planificación contingente en la que se pueden observar parcialmente los cambios producidos por las acciones durante su ejecución. Puede ser determinista y estocástica. Otro de los tipos de planificación es la planificación conformante que es una variación de la planificación clásica en la que el entorno no es observable y las transiciones pueden ser o no deterministas.

Como forma de resolución del proceso de planificación, se distinguen dos tipos en función de la dependencia o independencia del dominio. En la primera, el planificador contiene información propia del dominio mientras que en la segunda, no.

El enfoque utilizado en este proyecto se basa en la planificación clásica que busca la obtención de un conjunto de acciones que, ejecutadas de forma secuencial a partir de un estado inicial determinado, conformen el estado final objetivo de dicha planificación. Este tipo de planificación considera que se dispone del conocimiento total del entorno en el que se opera y que el resultado de las acciones ejecutadas es siempre el mismo, es decir, son deterministas. Puesto que en StarCraft hay incertidumbre, la solución por la que se ha optado es realizar la planificación de forma determinista, ejecutar el plan y, en caso de fallo, volver a planificar.

2.3.1. Modelo conceptual

La planificación clásica puede representarse formalmente mediante un sistema de transición de estados finitos y deterministas cuya información se conoce completamente. Este sistema puede representarse mediante la siguiente tupla:

$$\sum = (S, A, \rho)$$

Donde:

- S: es el conjunto de estados finitos del problema.
- A: es el conjunto finito de acciones deterministas.

- ρ: es la función de transición entre los estados de tal forma que ρ: SxA → S.

2.3.2. Lenguaje de representación

Para poder representar los elementos definidos en el modelo conceptual es necesario el uso de un lenguaje que permita expresar dichos elementos de forma que un planificador sea capaz de operar con ellos. El lenguaje de representación utilizado en este proyecto recibe la denominación de PDDL cuyo significado es lenguaje de definición de dominios de planificación. PDDL es un lenguaje de representación de alto nivel creado con la intención de estandarizar los distintos ya existentes en el campo de la inteligencia artificial. Fue desarrollado en 1998 por Drew McDermott principalmente para la competición internacional de planificación o IPC por sus siglas en inglés. A partir de ese momento, el lenguaje ha ido evolucionando incorporando nuevas funcionalidades para cada competición. (11).

Este lenguaje utiliza lógica de predicados para representar las acciones y los estados y permite la representación de las dos entradas de cualquier planificador independiente del dominio: dominio y problema. Los elementos que se definen en el dominio son:

- Tipos: la clasificación de los objetos.
- Predicados: las propiedades relacionadas con los objetos cuyos valores son booleanos.
- Funciones: permiten representar propiedades numéricas.
- Acciones: su definición contiene los parámetros, variables que pueden instanciarse con objetos de los distintos tipos declarados del dominio. También incluye una lista de condiciones que se han de cumplir para que la acción se pueda llevar a cabo y una lista de post-condiciones o efectos resultantes de su ejecución.

El fichero de problema contiene la definición sobre:

- Objetos: contiene las instancias de los tipos de objetos definidos en el dominio.
- Estado inicial: predicados y funciones sobre los objetos y el mundo que conforman el estado de partida del problema a resolver.
- Objetivos: predicados y funciones que se desea alcanzar.

En el fichero mostrado en la figura 5 se muestra un dominio sencillo que contiene la definición de dos tipos de objetos, base y juego. Sobre estos tipos de objetos se construyen los predicados que notifican si se ha alcanzado la población máxima, si se pueden crear trabajadores y si hay disponibles en la base. Como funciones se presentan los minerales o recursos disponibles, la población ocupada y la máxima población que se puede ocupar. Como acciones se describe la posibilidad de crear un trabajador que pasará a ser un trabajador desocupado si ha podido tomar un lugar en la población disponible y se disponen de los recursos necesarios, actualizando dichos valores tras su ejecución.

```
(define (domain StarCraft)
(:requirements :typing :fluents)
(:types Base Juego)
(:predicates
        (Poblacion_maxima ?j - Juego)
        (Puede_crear_unidades ?b - base)
        (Trabajadores_disponibles ?b -Base))

(:functions
        (minerales)
        (poblacion_actual)
        (poblacion_maxima)
        (trabajadores_desocupados))

(:action CrearTrabajadores
:parameters (?b - Base ?j -Juego)
:precondition (and
                (not (Poblacion_maxima ?j))
                (>= (minerales) 50)
                (Puede_crear_unidades ?b)
                (< (poblacion_actual) (poblacion_maxima)))

:effect (and
        (Trabajadores_disponibles ?b)
        (increase (trabajadores_desocupados) 1)
        (decrease (minerales) 50)
        (increase (poblacion_actual) 2))))
```

Figura 5: Ejemplo de definición de dominio en PDDL

En la figura 6 se muestra un problema a resolver, con un juego y dos bases aunque en solo una de ellas se puede crear trabajadores y unos recursos y valores de población. Al final del fichero se encuentran los objetivos a cumplir, disponer de cinco trabajadores desocupados y al menos 13 de población utilizada.

```
(define (problem problema1) (:domain StarCraft)
(:objects
        Base0 - Base
        Juego0 - Juego
        Base1 - Base)

(:init
        (Puede_crear_unidades Base1)
        (Trabajadores_disponibles Base1)
        (= (minerales) 50)
        (= (poblacion_actual) 8)
        (= (poblacion_maxima) 20)
        (= (trabajadores_desocupados) 4))

(:goal
        (and
                (= (trabajadores_desocupados) 5)
                (> (poblacion_actual) 12))))
```

Figura 6: Ejemplo de definición de estado en PDDL

2.3.3. Algoritmos

Los algoritmos que operan en planificación automática intentan solucionar el problema mediante la búsqueda de un camino en un grafo conformado por los estados como nodos y las acciones del dominio que se pueden ejecutar en ese estado como las transiciones. Estos algoritmos están compuestos por diferentes elementos que son descritos a continuación:

- Dirección: un algoritmo puede realizar su tarea partiendo desde el estado inicial indicado o bien desde el estado final deseado. A la primera de las aproximaciones se la denomina búsqueda hacia adelante y a la segunda, hacia atrás.

- Selección: la forma en la que se decide cuál es el próximo nodo a explorar. En un principio, se utilizaron algoritmos que realizaban la búsqueda en todo el espacio de problemas de forma metódica. Estos algoritmos son conocidos como algoritmos de fuerza bruta, ya que resuelven el problema comprobando cada posible solución al mismo. Estos algoritmos tenían como principal problema la elevada cantidad de tiempo que requerían para la obtención de una solución ya que, independientemente de cuál fuera el proceso seguido, anchura o profundidad, el espacio de búsqueda sobre el que podrían verse obligados a buscar seguía siendo elevado. Para solucionar este problema, se introdujo el uso de heurísticas que proporcionaban conocimiento sobre el problema y determinaban que nodos poseían las mejores opciones para ser explorados.

Estos algoritmos han sido utilizados en diferentes planificadores tales como FF (12). FF es un sistema de planificación heurística que estima las distancias al objetivo y considera que las acciones no son independientes entre sí. FF combina la búsqueda sistemática con el método de búsqueda local de escalada consistente en la búsqueda de una acción que reduzca el coste con respecto al padre. Un planificador que utiliza este algoritmo y que es utilizado en este proyecto es Metric-FF (13).

2.4. PELEA

Para utilizar la planificación en StarCraft es necesario el uso de una arquitectura que permita el control, la planificación y la ejecución de los planes generados. Este tipo de arquitecturas cuyo objetivo es la toma de decisiones sobre las acciones y la gestión de las tareas requeridas por las acciones recibe la denominación de arquitectura de control. Por ello, se ha elegido PELEA, que es una abreviación de "***P**lanning, **E**xecution and **LE**arning **A**rchitecture"* y es una arquitectura de control que permite realizar tareas de planificación, control y ejecución mediante un servicio web (8).

2.4.1. Arquitectura de PELEA

El diagrama ilustrativo de la arquitectura de PELEA se encuentra en la figura 7.

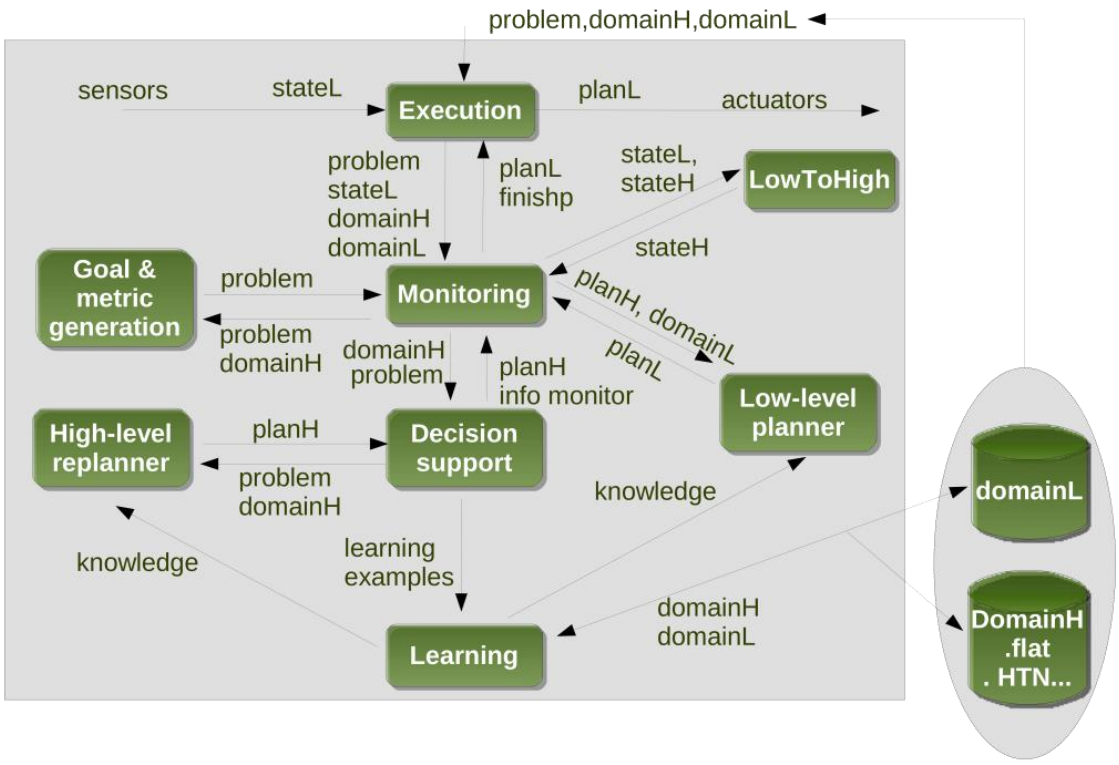

Figura 7: Arquitectura de PELEA

Los distintos componentes son:

- *Execution:* se encarga de la interacción de PELEA con el entorno, de inicializar PELEA cuando recibe un dominio y un problema, observar el entorno y de enviar la información de ejecución a los actuadores.

- *Monitoring:* se encarga del control de la ejecución del plan obtenido mediante el análisis de los objetivos en el estado que le vaya proporcionando el módulo *Execution.* También es el encargado de enviar las acciones a ejecutar.

- *Decision support:* se encarga de definir qué datos deben ser controlados por el módulo *monitoring* y de activar el proceso deliberativo cuando existan diferencias entre el estado esperado y el obtenido.

- *Hihg-level replanner:* es el encargado de generar un plan con un dominio y un problema. Además, permite la re-planificación sobre un plan parcialmente ejecutado.

- *Learning module:* genera conocimiento de control y dominio sobre los dos planificadores, alto y bajo nivel.

Y donde los mensajes se corresponden con:

- *stateL*: información del estado de bajo nivel procedente de los sensores.

- *stateH*: es el estado de alto nivel conformado por la información agregada o generalizada del *stateL*.

- *goals*: el grupo de objetivos del problema.

- *metric*: la métrica que se usa para la planificación de alto nivel.

- *planH:* conjunto de acciones de alto nivel que pueden ser secuenciales o paralelas y que pueden servir de objetivos para los planes de bajo nivel.

- *planL:* conjunto de acciones paralelas que se pueden operar directamente en el entorno.

- *domainH:* dominio con las acciones de alto nivel.

- *domainL:* dominio con las acciones de bajo nivel.

- *learning examples:* son utilizados para permitir el aprendizaje para futuras planificaciones.

- *heuristics:* distintos tipos de heurísticas para mejorar el rendimiento en la planificación.

- *monitoringInfo:* meta conocimiento que permite el seguimiento de la ejecución.

2.4.2. XPDDL

Las representaciones del dominio, estado y plan que utiliza PELEA se realizan en PDDL encapsulado dentro de un documento de tipo XML. Por eso el nombre que recibe es XPDDL. Para cada uno de los diferentes tipos de representaciones anteriormente mencionadas existe un documento *schema* con las características que lo describe. En la figura 8 se muestra una representación del *schema* del dominio y un ejemplo sencillo en XPDDL.

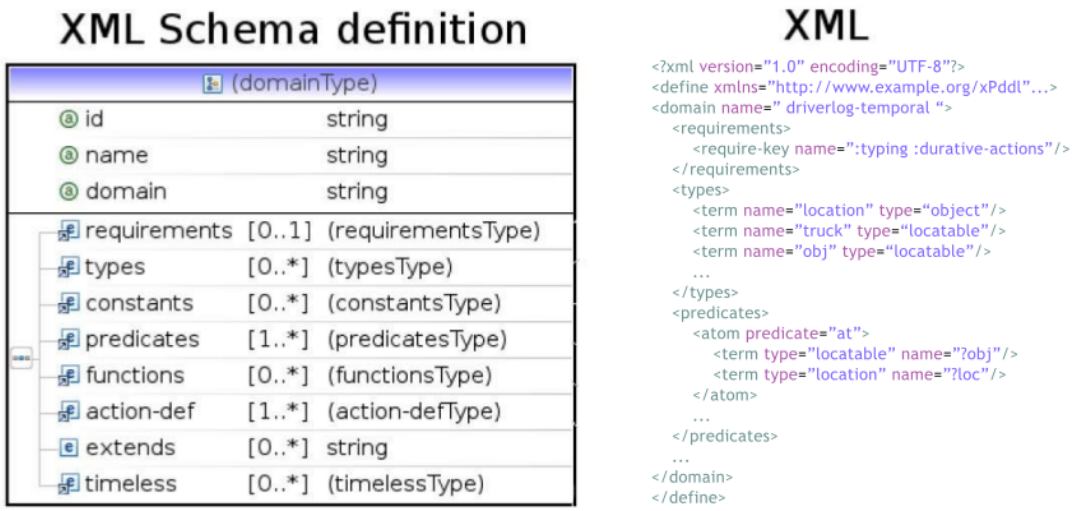

Figura 8: XPDDL

Capítulo 3: Diseño e implementación del sistema

En este capítulo se realiza una descripción del sistema de control que ha sido desarrollado. En el apartado 3.1 se presentan los casos de uso que han sido definidos para este proyecto. En el apartado 3.2 se describen los requisitos funcionales y no funcionales que han sido obtenidos a partir de los casos de uso. Posteriormente se describe el entorno operacional en el cuál se enmarca. A continuación en el apartado 3.4 se presenta el diseño arquitectónico del sistema mediante su descomposición en componentes, junto con la estructura de las acciones que han sido implementadas y los diagramas de flujo que describen el comportamiento del sistema.

3.1. Casos de Uso

Se procede a describir los casos de uso del sistema. Por claridad se encuentra dividido en las tres figuras siendo la primera de ellas (figura 9) una visión general de los casos de uso, la segunda (figura 10), el listado de acciones ejecutables y la tercera (figura 11) las acciones que puede realizar un usuario sobre el sistema.

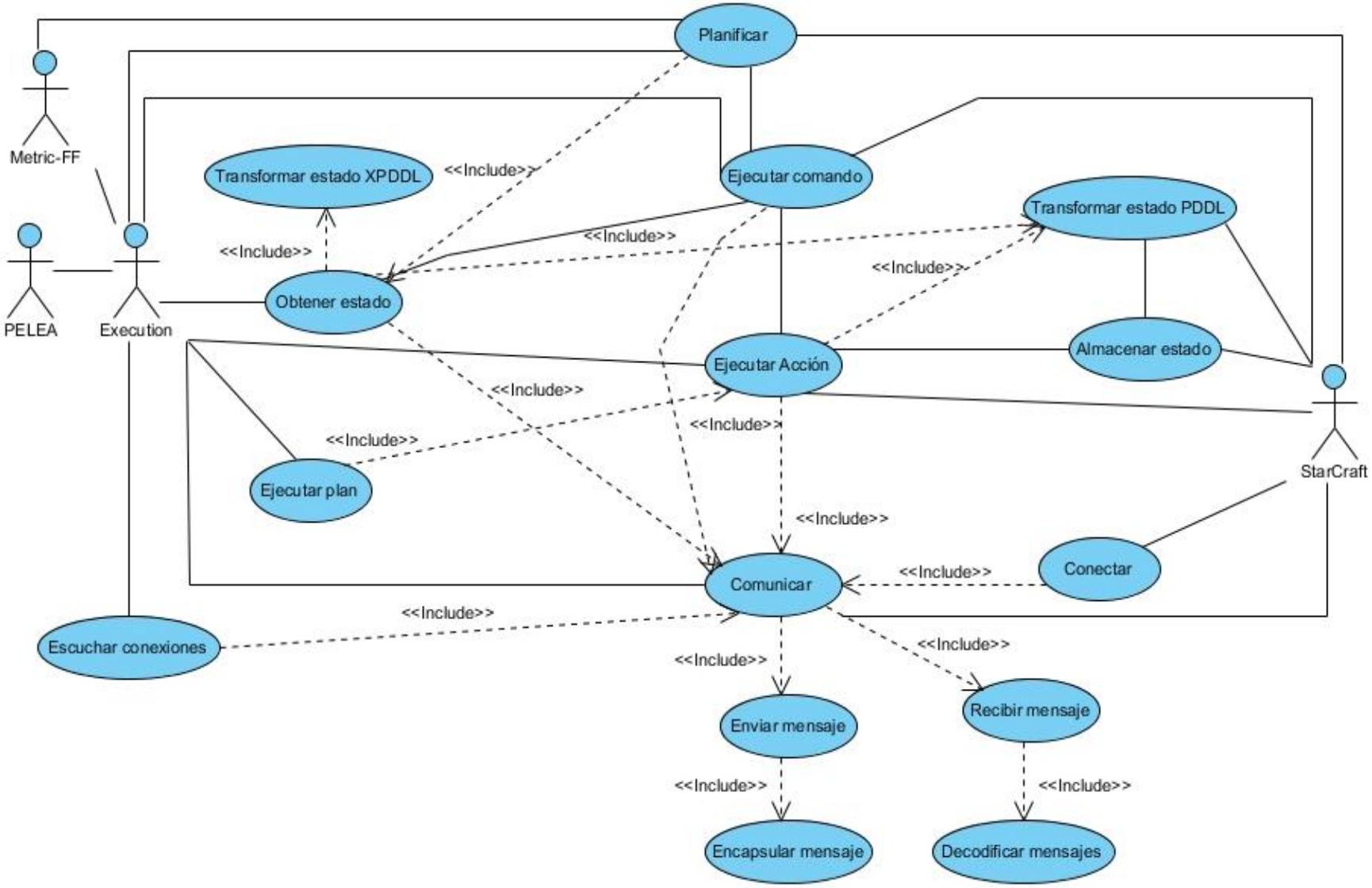

Figura 9: Diagrama de casos de uso general

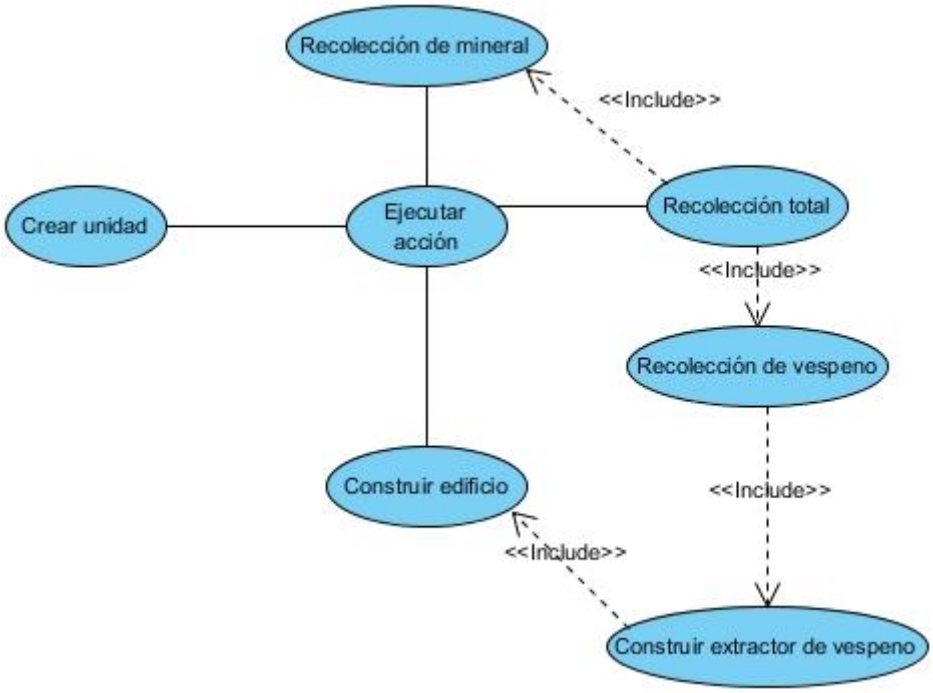

Figura 10: Diagrama de casos de uso de acciones.

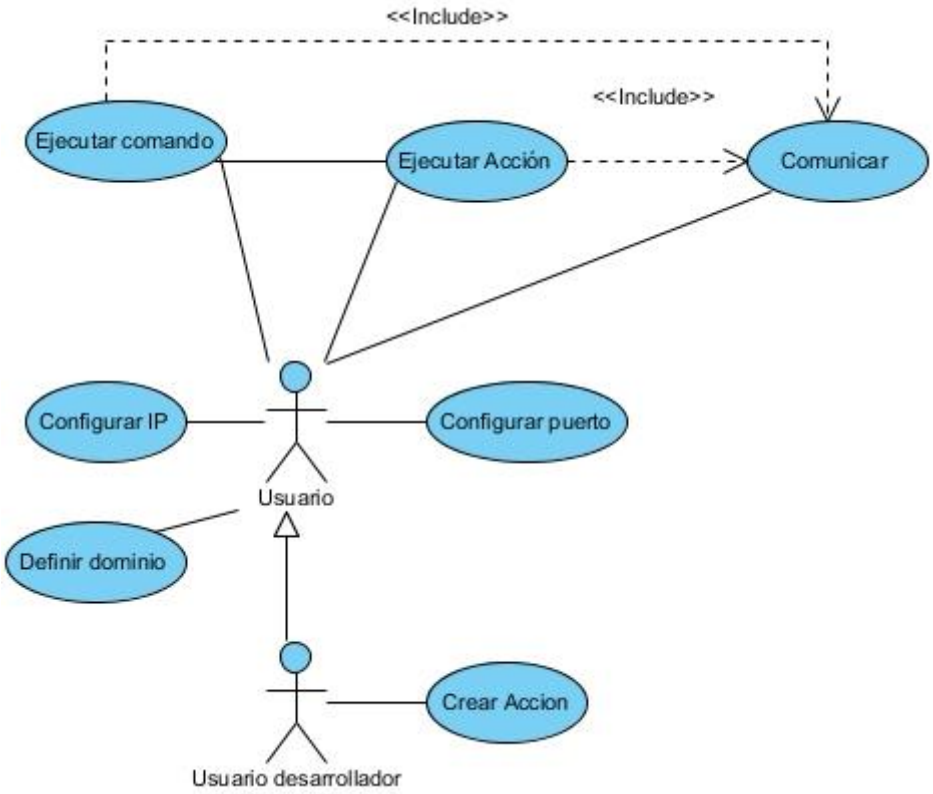

Figura 11: Diagrama de casos de uso para los usuarios.

Siendo los actores:

1. Metric-FF: el planificador que el sistema utiliza para generar los planes.
2. PELEA: el entorno de control para la planificación.
3. Execution: el módulo de PELEA que permite la ejecución, planificación y comunicación con el sistema.
4. StarCraft: el juego sobre el cuál se va a aplicar los comportamientos, StarCraft.
5. Usuario: persona encargada de ejecutar el sistema y controlar su funcionamiento.
6. Usuario desarrollador: persona encargada de crear los comportamientos codificados.

3.1.1. Descripción de los atributos de los casos de uso

El formato que se sigue para la descripción textual de los casos de uso es el siguiente:

- Nombre: el nombre que recibe el caso de uso.
- ID: el identificador único del caso de uso con el formato CU-XX donde XX es un número.
- Descripción: breve descripción del caso de uso.
- Precondiciones: las condiciones que se han de cumplir para poder efectuar el caso de uso.
- Secuencia básica: el escenario normal de desarrollo.
- Secuencia alternativa: otros posibles escenarios del caso de uso.
- Postcondiciones: los efectos que se obtienen tras efectuar el caso de uso.

3.1.2. Descripción textual de los casos de uso

La descripción de los casos de uso que se mostraron en las figuras del apartado 3.1 son:

Nombre	Planificar	**ID**	CU-01
Descripción	Se solicita al programa realizar una planificación.		
Actores	StarCraft, *Execution*, PELEA, Metric-FF, Usuario.		
Precondiciones	• Se ha establecido una conexión. • Se dispone de un dominio en un fichero PDDL. • Se dispone de Metric-FF compilado.		
Secuencia básica	1. PELEA solicita a través de *Execution* la información del estado. 2. El sistema envía el mensaje para la obtención del estado. 3. Se decodifica el mensaje de envío de estado. 4. Se transforma el estado almacenado a PDDL. 5. Se envía el estado en PDDL. 6. Se recibe el estado y se transforma a XPDDL. 7. Se cede el estado a *Execution*.		
Secuencia alternativa	1. El usuario o el sistema solicita una planificación. 2. Se transforma el estado a PDDL. 3. Se envía el estado del juego en PDDL. 4. Se almacena el estado en un fichero en PDDL. 5. Se envía el comando para realizar la planificación. 6. Se realiza la llamada a Metric-FF para la planificación. 7. Se decodifica la salida del planificador.		
Postcondiciones	• Se dispone de un plan para el estado.		

Tabla 1: Descripción del caso de uso CU-01

Nombre	Transformar el estado a XPDDL	**ID**	CU-02
Descripción	El sistema recibe un estado en PDDL que transforma a XPDDL para su uso en PELEA.		
Actores	PELEA, *Execution*.		
Precondiciones	• Se dispone de un estado en PDDL.		
Secuencia básica	1. Se guarda en un fichero el estado en PDDL. 2. Se llama al transformador de PELEA para realizar la transformación.		
Secuencia alternativa	No hay secuencia alternativa.		
Postcondiciones	• Estado en XPDDL.		

Tabla 2: Descripción del caso de uso CU-02

Nombre	Ejecutar comando	**ID**	CU-03
Descripción	Se solicita la ejecución de un comando al sistema.		
Actores	Usuario, PELEA, *Execution*, StarCraft.		
Precondiciones	• El sistema está funcionando.		
Secuencia básica	1. El usuario, PELEA o StarCraft envían un comando. 2. El sistema decodifica el mensaje. 3. Se ejecuta el comando ordenado.		
Secuencia alternativa	1. El usuario, PELEA o StarCraft envían un comando. 2. El sistema decodifica el mensaje. 3. Se envía el comando. 4. Se identifica el comando recibido. 5. Se ejecuta el comando ordenado.		
Postcondiciones	• El comando ejecutado.		

Tabla 3: Descripción del caso de uso CU-03

Nombre	Transformar estado a PDDL	**ID**	CU-04
Descripción	Se transforma y complementa el estado almacenado con la información del juego en PDDL.		
Actores	StarCraft, PELEA, Usuario.		
Precondiciones	• Se ha establecido una conexión. • Se dispone de un estado almacenado.		
Secuencia básica	1. El usuario o PELEA ordenan la obtención del estado. 2. Se transforma el estado almacenado a PDDL. 3. Se complementa el estado con la información obtenida del juego en PDDL.		
Secuencia alternativa	No hay secuencia alternativa.		
Postcondiciones	• El estado en formato PDDL.		

Tabla 4: Descripción del caso de uso CU-04

Nombre	Obtener el estado	**ID**	CU-05
Descripción	Se solicita al sistema el estado en formato PDDL.		
Actores	PELEA, *Execution,* Usuario, StarCraft.		
Precondiciones	• Se dispone de un estado almacenado. • Se ha establecido una conexión.		
Secuencia básica	1. PELEA a través de *Execution* solicita el estado. 2. Se envía el comando para obtener el estado. 3. Se interpreta el comando para obtener el estado. 4. Se transforma el estado a PDDL. 5. Se envía el estado. 6. Se decodifica el mensaje y se extrae el estado. 7. Se transforma el estado a XPDDL y se entrega a *Execution* quien se lo envía a PELEA.		
Secuencia alternativa	1. El usuario solicita el estado. 2. Se envía el comando para obtener el estado. 3. Se interpreta el comando para obtener el estado. 4. Se transforma el estado a PDDL. 5. Se envía el estado. 6. Se decodifica el mensaje y se extrae el estado. 7. Se almacena el estado en un fichero en PDDL.		
Postcondiciones	• Se dispone del estado en PDDL en PELEA y fichero.		

Tabla 5: Descripción del caso de uso CU-05

Nombre	Ejecutar acción	**ID**	CU-06
Descripción	Se solicita al sistema la ejecución de una acción en PDDL.		
Actores	PELEA, *Execution,* Usuario, StarCraft.		
Precondiciones	• Se dispone de una acción en PDDL o de un plan. • Se ha establecido conexión.		
Secuencia básica	1. PELEA a través de *Execution* envía una acción para ser ejecutada. 2. El sistema envía codificada la acción. 3. Se decodifica la acción y se añade a la lista de acciones para ejecutar. 4. Se ejecuta la acción sobre StarCraft. 5. Se modifica el estado con los resultados de la acción.		
Secuencia alternativa	1. El usuario solicita la ejecución de una acción. 2. El sistema envía codificada la acción. 3. Se decodifica la acción y se añade a la lista de acciones para ejecutar. 4. Se ejecuta la acción sobre StarCraft. 5. Se modifica el estado con los resultados de la acción.		
Postcondiciones	• La acción ha sido ejecutada en caso de ser posible y sus resultados han sido almacenados en el estado.		

Tabla 6: Descripción del caso de uso CU-06

Nombre	Almacenar estado	**ID**	CU-07
Descripción	El sistema almacena una representación del juego.		
Actores	StarCraft.		
Precondiciones	• El sistema está funcionando.		
Secuencia básica	1. El juego indica mediante eventos la información a actualizar. 2. Se analiza la información de los eventos. 3. Se almacena en el estado.		
Secuencia alternativa	1. Tras la ejecución de una acción se realizan cambios en el estado.		
Postcondiciones	• El estado almacenado se actualiza.		

Tabla 7: Descripción del caso de uso CU-07

Nombre	Ejecutar plan	**ID**	CU-08
Descripción	Se ejecuta un plan obtenido del planificador.		
Actores	PELEA, *Execution,* Usuario, StarCraft.		
Precondiciones	• Se ha establecido conexión. • Se dispone de un plan en PDDL.		
Secuencia básica	1. PELEA a través de *Execution* solicita la ejecución de una acción. 2. El sistema envía la acción codificada para su ejecución. 3. Se decodifica y ejecuta la acción. 4. Se devuelve un mensaje con el resultado de la ejecución. 5. Se repiten los pasos 2,3 y 4 hasta la finalización del plan.		
Secuencia alternativa	1. El sistema dispone de un plan almacenado. 2. El sistema envía la acción codificada para su ejecución. 3. Se decodifica y ejecuta la acción. 4. Se devuelve un mensaje con el resultado de la ejecución. 5. Se repiten los pasos 2,3 y 4 hasta la finalización del plan.		
Postcondiciones	• El plan se ha ejecutado en el juego.		

Tabla 8: Descripción del caso de uso CU-08

Nombre	Comunicar	**ID**	CU-09
Descripción	Se realiza el envío de mensajes entre las partes del sistema.		
Actores	PELEA, *Execution,* Usuario, StarCraft.		
Precondiciones	• Se ha establecido una conexión.		
Secuencia básica	1. StarCraft, el usuario o PELEA solicitan el envío de un mensaje. 2. El mensaje es codificado y enviado.		
Secuencia alternativa	1. StarCraft, el usuario o PELEA solicitan la recepción de un mensaje. 2. El mensaje es recibido y decodificado.		
Postcondiciones	• La comunicación entre ambas partes se ha resuelto.		

Tabla 9: Descripción del caso de uso CU-09

Nombre	Conectar	**ID**	CU-010
Descripción	StarCraft solicita una conexión.		
Actores	PELEA, *Execution*, StarCraft.		
Precondiciones	• El sistema está en funcionamiento. • *Execution* está en escucha.		
Secuencia básica	1. StarCraft solicita la conexión. 2. *Execution* acepta la conexión.		
Secuencia alternativa	No hay secuencia alternativa.		
Postcondiciones	• La conexión se ha establecido.		

Tabla 10: Descripción del caso de uso CU-010

Nombre	Escuchar conexión	**ID**	CU-011
Descripción	*Execution* se mantiene a la escucha para nuevas conexiones.		
Actores	*Execution.*		
Precondiciones	• El sistema está activo.		
Secuencia básica	1. *Execution* se mantiene a la espera de nuevas conexiones.		
Secuencia alternativa	No hay secuencia alternativa.		
Postcondiciones	• El sistema está preparado para aceptar nuevas conexiones.		

Tabla 11: Descripción del caso de uso CU-011

Nombre	Enviar mensaje	**ID**	CU-012
Descripción	El sistema envía un mensaje.		
Actores	PELEA, *Execution,* Usuario, StarCraft.		
Precondiciones	• Se ha establecido una conexión.		
Secuencia básica	1. El usuario, PELEA a través de *Execution* o StarCraft solicitan el envío de un mensaje. 2. El mensaje es encapsulado y enviado.		
Secuencia alternativa	No hay secuencia alternativa.		
Postcondiciones	• El mensaje ha sido enviado.		

Tabla 12: Descripción del caso de uso CU-012

Nombre	Recibir mensaje	**ID**	CU-013
Descripción	El sistema recibe un mensaje		
Actores	*Execution,* StarCraft.		
Precondiciones	• Se ha establecido una conexión. • Se ha enviado un mensaje.		
Secuencia básica	1. El sistema recibe un mensaje. 2. Se decodifica el mensaje.		
Secuencia alternativa	No hay secuencia alternativa.		
Postcondiciones	• El mensaje se encuentra disponible en el destino.		

Tabla 13: Descripción del caso de uso CU-013

Nombre	Encapsular mensaje	ID	CU-014
Descripción	El sistema prepara un mensaje para su envío.		
Actores	*Execution,* StarCraft.		
Precondiciones	• Se dispone de un mensaje.		
Secuencia básica	1. El mensaje se encapsula en las cabeceras para el envío del mensaje.		
Secuencia alternativa	No hay secuencia alternativa.		
Postcondiciones	• El mensaje está listo para el envío.		

Tabla 14: Descripción del caso de uso CU-014

Nombre	Decodificar mensaje	ID	CU-015
Descripción	El mensaje recibido se decodifica para su uso.		
Actores	*Execution,* StarCraft.		
Precondiciones	• Se ha recibido un mensaje.		
Secuencia básica	1. Se analiza y decodifica el mensaje para su uso.		
Secuencia alternativa	No hay secuencia alternativa.		
Postcondiciones	• Se dispone del mensaje preparado para su uso.		

Tabla 15: Descripción del caso de uso CU-015

Nombre	Recolección de mineral	ID	CU-016
Descripción	Se ordena la acción que permite que un trabajador empiece la extracción de mineral.		
Actores	StarCraft.		
Precondiciones	• Se ha ordenado la ejecución de la acción.		
Secuencia básica	1. Se analiza los recursos de la base y los trabajadores. 2. Se envían los trabajadores de StarCraft justos para una recolección de minerales. 3. Se dan de alta los efectos de la acción.		
Secuencia alternativa	1. Se analizan los recursos de la base y los trabajadores. 2. Se envían los trabajadores de StarCraft disponibles para una recolección de minerales. 3. Se notifica que no se ha podido cumplir al completo la acción.		
Postcondiciones	• Los trabajadores disponibles empiezan la recolección de minerales.		

Tabla 16: Descripción del caso de uso CU-016

Nombre	Crear unidad	ID	CU-017
Descripción	Se ordena al sistema la creación de una unidad.		
Actores	StarCraft.		
Precondiciones	• Se ha ordenado la ejecución de la acción. • Se dispone del edificio para su creación.		
Secuencia básica	1. Se analizan los recursos de la base en función de la unidad a crear. 2. Se identifica el edificio capaz de crearla. 3. Se ordena su creación a StarCraft. 4. Se da de alta la unidad en el estado.		
Secuencia alternativa	1. Se analizan los recursos de la base en función de la unidad a crear. 2. Se espera hasta que se disponga de los recursos para la creación.		
Postcondiciones	• La unidad ordenada se crea.		

Tabla 17: Descripción del caso de uso CU-017

Nombre	Recolección total	ID	CU-018
Descripción	Se ordena comprobar si los recursos están siendo explotados.		
Actores	StarCraft.		
Precondiciones	• Se ha ordenado la ejecución de la acción.		
Secuencia básica	1. Se analiza los recursos de la base y los trabajadores asociados a su extracción. 2. Se da de alta en el estado el resultado.		
Secuencia alternativa	No hay secuencia alternativa.		
Postcondiciones	• Si se están explotando los recursos esta información se almacena en el estado.		

Tabla 18: Descripción del caso de uso CU-018

Nombre	Recolección de vespeno	ID	CU-019
Descripción	Se ordena la acción que permite que un trabajador empiece la extracción de vespeno.		
Actores	StarCraft.		
Precondiciones	• Se ha ordenado la ejecución de la acción. • Se dispone de un extractor de vespeno.		
Secuencia básica	1. Se analizan los recursos de la base y los trabajadores. 2. Se envían los trabajadores de StarCraft justos para una recolección de vespeno. 3. Se da de alta los efectos de la acción.		
Secuencia alternativa	1. Se analizan los recursos de la base y los trabajadores. 2. Se envían los trabajadores de StarCraft disponibles para una recolección de vespeno en caso de disponerse de un extractor. 3. Se notifica que no se ha podido cumplir al completo la acción.		
Postcondiciones	• Los trabajadores disponibles empiezan la recolección de vespeno.		

Tabla 19: Descripción del caso de uso CU-019

Nombre	Construir edificio	ID	CU-020
Descripción	Se ordena al sistema la ejecución de la acción para la construcción de un edificio.		
Actores	StarCraft.		
Precondiciones	• Se ha ordenado la construcción del edificio. • Se cumplen los requisitos tecnológicos para su construcción.		
Secuencia básica	1. Se analizan los recursos y trabajadores. 2. Se identifica un emplazamiento. 3. Se ordena a StarCraft la construcción del edificio. 4. Se almacena en el estado el edificio construido.		
Secuencia alternativa	1. En caso de no cumplir algún requisito se notifica que no se ha podido llevar a cabo.		
Postcondiciones	• Se dispone del edificio ordenado construir.		

Tabla 20: Descripción del caso de uso CU-020

Nombre	Construir extractor de vespeno	ID	CU-021
Descripción	Se ordena al sistema la ejecución de la acción para la construcción de un extractor de vespeno.		
Actores	StarCraft.		
Precondiciones	• Se ha ordenado la construcción del edificio. • Se dispone de un géiser para la construcción.		
Secuencia básica	1. Se analizan los recursos y trabajadores de la base. 2. Se realiza la construcción del edificio. 3. Se almacena en el estado que se dispone de extractor.		
Secuencia alternativa	1. En caso de no cumplir algún requisito se notifica que no se ha podido llevar a cabo.		
Postcondiciones	• Se dispone de un extractor para la recolección de vespeno.		

Tabla 21: Descripción del caso de uso CU-021

Nombre	Configurar IP	ID	CU-022
Descripción	Se establece la dirección IP para las comunicaciones.		
Actores	Usuario.		
Precondiciones	• Se dispone de un fichero de configuración.		
Secuencia básica	1. Se dispone de un fichero para la configuración. 2. Se introduce un campo en el fichero de configuración con la IP deseada.		
Secuencia alternativa	No hay secuencia alternativa.		
Postcondiciones	• La IP para las conexiones se actualiza.		

Tabla 22: Descripción del caso de uso CU-022

Nombre	Configurar puerto	ID	CU-023
Descripción	Se establece el puerto para las comunicaciones.		
Actores	Usuario.		
Precondiciones	• Se dispone de un fichero de configuración.		
Secuencia básica	1. Se dispone de un fichero para la configuración. 2. Se introduce un campo en el fichero de configuración con el puerto deseado.		
Secuencia alternativa	No hay secuencia alternativa.		
Postcondiciones	• El puerto para las conexiones se actualiza.		

Tabla 23: Descripción del caso de uso CU-023

Nombre	Definir dominio	ID	CU-024
Descripción	Se define el dominio en el cual opera el *bot*.		
Actores	Usuario.		
Precondiciones	• Se dispone de un fichero para el dominio.		
Secuencia básica	1. El usuario abre el fichero en PDDL del dominio. 2. El usuario modifica el dominio.		
Secuencia alternativa	No hay secuencia alternativa.		
Postcondiciones	• El dominio queda definido.		

Tabla 24: Descripción del caso de uso CU-024

Nombre	Crear acción	ID	CU-025
Descripción	El usuario desarrollador introduce en el sistema una nueva acción o comando.		
Actores	Usuario desarrollador.		
Precondiciones	• Se dispone del código fuente.		
Secuencia básica	1. El usuario define un nuevo comportamiento para una acción o comando. 2. Se compila y despliega el nuevo código.		
Secuencia alternativa	No hay secuencia alternativa.		
Postcondiciones	• El sistema dispone de nuevos comandos y acciones.		

Tabla 25: Descripción del caso de uso CU-025

3.2. Requisitos del Sistema

En este apartado se especifican los requisitos funcionales y no funcionales que han sido identificados en la fase de análisis.

3.2.1. Descripción de los atributos de los requisitos

Los requisitos estarán contenidos en una tabla con el siguiente formato:

- ID: código identificador del requisito con formato TIPO-XX donde XX es un número.
  - Tipo: indica el tipo del requisito.
    - *FUN:* requisitos funcionales.
    - *NFUN:* requisitos no funcionales.
- Nombre: nombre descriptivo y resumen del requisito.
- Descripción: breve descripción del requisito.
- Necesidad: cómo de necesario es el requisito, pudiendo ser:
  - Esencial: se requiere de forma categórica el requisito para el propósito del sistema.
  - Deseable: el requisito se considera necesario pero no vital.
  - Opcional: la necesidad del requisito es baja.
- Prioridad: la importancia del requisito pudiendo ser:
  - Alta: el requisito tiene la mayor consideración.
  - Media: el requisito es de una importancia moderada.
  - Baja: la importancia del requisito está por debajo de la media.
- Estabilidad: Cómo de estable es un requisito. Sus posibles valores son:
  - Estable: el requisito no se modificará.
  - No estable: es posible que el requisito sufra modificaciones.
- Claridad: Indica si el requisito es claro.
  - Alta: la descripción del requisito es precisa y minuciosa.
  - Media: la descripción del requisito es precisa.
  - Baja: el requisito es una guía del comportamiento deseado.

- Fuente: Indica el número o números de los casos de uso de los cuáles procede el requisito. En caso de proceder directamente del cliente, se especifica con las siglas "CLI".

3.2.2. Requisitos funcionales

Nombre	PELEA			**ID**	FUN-01
Descripción	El sistema hará uso de PELEA para la generación y control de ejecución de planes.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-01, CU-08.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 26: requisito FUN-01

Nombre	Planificador			**ID**	FUN-02
Descripción	Se requerirá del uso del planificador Metric-FF compilado para la plataforma sobre la cual PELEA opere.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-01.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 27: requisito FUN-02

Nombre	Servidor			**ID**	FUN-03
Descripción	Se generará un servidor reducido que permita interactuar con PELEA así como comunicarse con el cliente del *bot* de StarCraft.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-09, CU-10, CU-11.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 28: requisito FUN-03

Nombre	Conexión servidor - PELEA			**ID**	FUN-04
Descripción	El servidor se comunicará con PELEA a través del módulo *Execution* del mismo.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-09.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 29: requisito FUN-04

Nombre	Independencia de plataforma del servidor			**ID**	FUN-05
Descripción	El código del servidor deberá ser capaz de ser ejecutado tanto en Windows como en Linux.				
Necesidad	Deseable.	**Prioridad**	Baja.	**Fuente**	CLI.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 30: requisito FUN-05

Nombre	Tratamiento de mensajes		ID	FUN-06	
Descripción	El servidor será capaz de tratar los mensajes enviados desde el cliente y almacenar el contenido de los mismos.				
Necesidad	Esencial.	Prioridad	Alta.	Fuente	CU-09, CU-13, CU-15.
Estabilidad	Estable.	Claridad	Alta.		

Tabla 31: requisito FUN-06

Nombre	Envío de mensajes		ID	FUN-07	
Descripción	El servidor será capaz de enviar los mensajes generados por PELEA con el formato requerido para que el cliente de StarCraft pueda interpretarlos.				
Necesidad	Esencial.	Prioridad	Alta.	Fuente	CU-09, CU-12, CU-14.
Estabilidad	Estable.	Claridad	Alta.		

Tabla 32: requisito FUN-07

Nombre	Paralelismo de comunicaciones		ID	FUN-08	
Descripción	El servidor operará en su propio hilo de ejecución para no interferir el correcto funcionamiento del módulo de PELEA al que se comunica.				
Necesidad	Deseable.	Prioridad	Alta.	Fuente	CU-11.
Estabilidad	Estable.	Claridad	Alta.		

Tabla 33: requisito FUN-08

Nombre	Existencia propia		ID	FUN-09	
Descripción	El servidor será capaz de operar con funcionalidad reducida sin hacer uso de PELEA.				
Necesidad	Deseable.	Prioridad	Media.	Fuente	CU-01, CU-03, CU-05, CU-08, CU-09, CU-11.
Estabilidad	Estable.	Claridad	Alta.		

Tabla 34: requisito FUN-09

Nombre	Puerto de escucha		ID	FUN-10	
Descripción	El servidor será capaz de mantenerse a la escucha en el puerto que se le indique.				
Necesidad	Esencial.	Prioridad	Alta.	Fuente	11,23.
Estabilidad	Estable.	Claridad	Alta.		

Tabla 35: requisito FUN-10

Nombre	Pérdida de conexión		**ID**	FUN-11	
Descripción	El servidor será capaz de mantenerse a la escucha de nuevo en el mismo puerto tras perder la comunicación con el cliente.				
Necesidad	Opcional.	**Prioridad**	Media.	**Fuente**	CU-11, CU-23.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 36: requisito FUN-11

Nombre	Solicitud de estado		**ID**	FUN-12	
Descripción	El servidor deberá ser capaz de solicitar al cliente el estado actual y realizar las transformaciones pertinentes para que PELEA pueda hacer uso de él.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-02, CU-04, CU-05, CU-07.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 37: requisito FUN-12

Nombre	Cliente		**ID**	FUN-13	
Descripción	Se creará un cliente capaz de comunicarse con el servidor que contiene PELEA.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-09.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 38: requisito FUN-13

Nombre	Almacenamiento del estado		**ID**	FUN-14	
Descripción	El cliente deberá ser capaz de almacenar una representación del estado del juego.				
Necesidad	Deseable.	**Prioridad**	Alta.	**Fuente**	CU-07.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 39: requisito FUN-14

Nombre	Tratamiento de mensajes		**ID**	FUN-15	
Descripción	El cliente deberá ser capaz de recibir, aislar y ejecutar los mensajes enviados por el servidor.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-09, CU-13, CU-15.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 40: requisito FUN-15

Nombre	Solicitud de conexión		**ID**	FUN-16	
Descripción	El cliente deberá ser capaz de conectarse con el servidor de PELEA a la dirección IP y puerto que se le indique.				
Necesidad	Deseable.	**Prioridad**	Alta.	**Fuente**	CU-10, CU-22, CU-23.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 41: requisito FUN-16

Nombre	Existencia propia		**ID**	FUN-17	
Descripción	El cliente deberá ser capaz de ofrecer una funcionalidad reducida aun cuando no haya podido realizar una conexión con el servidor.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-03, CU-04, CU-07.
Estabilidad	Estable.	**Claridad**	Media.		

Tabla 42: requisito FUN-17

Nombre	Control de objetos		**ID**	FUN-18	
Descripción	El cliente deberá de ser capaz de ofrecer a las acciones los objetos que el *bot* haya registrado para su uso en las acciones programadas.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-06, CU-07.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 43: requisito FUN-18

Nombre	Tipos de objetos		**ID**	FUN-19	
Descripción	El cliente deberá ser capaz de almacenar cualquier tipo de objeto definido por el usuario.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-07.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 44: requisito FUN-19

Nombre	Solicitud de actualización de valores		**ID**	FUN-20	
Descripción	El cliente deberá ser capaz de solicitar a los sensores del usuario la actualización de valores de funciones.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-05.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 45: requisito FUN-20

Nombre	Forma de acciones		**ID**	FUN-21	
Descripción	El cliente deberá ser capaz de ejecutar acciones programadas en clases así como implementadas mediante *callbacks*				
Necesidad	Deseable.	**Prioridad**	Alta.	**Fuente**	CU-06, CU-25.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 46: requisito FUN-21

Nombre	Registro de sucesos (*log*)		**ID**	FUN-22	
Descripción	El cliente deberá ser capaz almacenar en un fichero de texto un registro con los sucesos acontecidos durante una ejecución del juego.				
Necesidad	Opcional.	**Prioridad**	Baja.	**Fuente**	CLI.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 47: requisito FUN-22

Nombre	Acciones básicas		**ID**	FUN-23	
Descripción	Se crearán un grupo de acciones con scripts generales que permitan el control de unidades dentro del juego.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-06, CU-16, CU-17, CU-18, CU-19, CU-20, CU-21
Estabilidad	Estable.	**Claridad**	Media.		

Tabla 48: requisito FUN-23

Nombre	*Script* de construcción		**ID**	FUN-24	
Descripción	Se generará un *script* básico genérico que permita la construcción de edificios de la raza Terran.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-20.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 49: requisito FUN-24

Nombre	*Script* de entrenamiento		**ID**	FUN-25	
Descripción	Se creará un *script* capaz de entrenar una unidad Terran que el *bot* requiera.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-17.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 50: requisito FUN-25

Nombre	*Script* de extracción de recursos		**ID**	FUN-26	
Descripción	Se implementarán dos *scripts* que permita a los trabajadores realizar la recolección de recursos, uno para vespeno y otro para minerales.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-19, CU-21.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 51: requisito FUN-26

Nombre	Acción genérica			**ID**	FUN-27
Descripción	Se generará una acción genérica capaz de realizar cambios en la representación interna del estado que enlace los objetos que reciba por parámetros a las postcondiciones que se le indiquen.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-06, CU-07.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 52: requisito FUN-27

Nombre	Constructor de extractor			**ID**	FUN-28
Descripción	Se generará un script capaz de construir el extractor de vespeno				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-19, CU-21.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 53: requisito FUN-28

Nombre	Base			**ID**	FUN-29
Descripción	El sistema deberá ser capaz de gestionar una abstracción no existente en el juego: base.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-16, CU-17, CU-18, CU-19, CU-20, CU-21.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 54: requisito FUN-29

Nombre	Procesos			**ID**	FUN-30
Descripción	El cliente y el servidor operarán en procesos diferentes.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-01, CU-09.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 55: requisito FUN-30

Nombre	Inicio de comunicaciones			**ID**	FUN-31
Descripción	El proceso en el que se ubique el cliente deberá ser capaz e iniciar la conexión al proceso del servidor.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-10.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 56: requisito FUN-31

3.2.3. Requisitos no funcionales

Nombre	Configuración de PELEA		**ID**	NFUN-01	
Descripción	Se generará un fichero de configuración XML habilitando los módulos de PELEA utilizados: *Execution*, *DecisionSupport* y *Monitoring*.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-22, CU-23.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 57: requisito NFUN-01

Nombre	Formato del estado		**ID**	NFUN-02	
Descripción	Se entregará a PELEA la representación del problema en PDDL transformado a su formato interno, XPDDL.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-02.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 58: requisito NFUN-02

Nombre	Dominio		**ID**	NFUN-03	
Descripción	Se generará un fichero en formato PDDL que contenga el dominio sobre el cuál el *bot* pueda operar.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-24.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 59: requisito NFUN-03

Nombre	Lenguaje del *bot*		**ID**	NFUN-04	
Descripción	El lenguaje de desarrollo del *bot* será C++.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CLI.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 60: requisito NFUN-04

Nombre	Ejecutable del *bot*		**ID**	NFUN-05	
Descripción	El ejecutable del *bot* será generado como una DLL.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CLI.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 61: requisito NFUN-05

Nombre	Sistema operativo		**ID**	NFUN-06	
Descripción	El sistema operativo en el cuál se ejecutará el *bot* será Microsoft Windows XP.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CLI.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 62: requisito NFUN-06

Nombre	Lenguaje del servidor		**ID**	NFUN-07	
Descripción	El servidor deberá ser programado en el lenguaje Java.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CLI.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 63: requisito NFUN-07

Nombre	Envío del estado		**ID**	NFUN-08	
Descripción	El cliente deberá ser capaz de enviar el estado del juego en formato PDDL cuando el servidor se lo requiera.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-04, CU-05.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 64: requisito NFUN-08

Nombre	Dominio		**ID**	NFUN-9	
Descripción	Se generará un dominio capaz de representar la interacción de las acciones básicas.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-24.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 65: requisito NFUN-09

Nombre	Funciones del dominio		**ID**	NFUN-10	
Descripción	El dominio generado deberá recoger valores numéricos en funciones.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-24.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 66: requisito NFUN-10

Nombre	No ralentización		**ID**	NFUN-11	
Descripción	El cliente de StarCraft deberá realizar su función de forma que no genere un retraso considerable en el juego.				
Necesidad	Esencial.	**Prioridad**	Alta.	**Fuente**	CU-09.
Estabilidad	Estable.	**Claridad**	Alta.		

Tabla 67: requisito NFUN-11

3.2.4. Funcionalidades

La funcionalidad del sistema es proporcionar una arquitectura para el desarrollo de *bots* capaces de interactuar con StarCraft, PELEA, BWAPI (14) y ChaosLauncher (15). Para ello, el sistema es capaz de mantener comunicaciones entre distintas máquinas que contengan cada uno de los componentes necesarios para su funcionamiento. Además, proporciona un grupo de clases definido para el desarrollo de *bots* además de ofrecer herramientas para el almacenaje sencillo de los datos referentes al estado, facilitando además su traducción a PDDL. Entre estas funcionalidades generales se destaca:

- La posibilidad de conectar con diferentes máquinas siempre que se encuentren conectadas y se disponga de su IP y puerto.
- La capacidad del servidor de funcionar independientemente de la plataforma.
- La nomenclatura automática de los objetos en función de su tipo.
- La capacidad de ofrecer a las acciones programadas los objetos definidos como parámetros en la acción del dominio.
- La flexibilidad para notificar los cambios de estado y la capacidad para ser llamados en el momento requerido.
- La traducción automática del estado a formato PDDL.

Se incluye también una serie de acciones básicas implementadas que pueden servir de base para la construcción de comportamientos complejos. Dichas funcionalidades son:

- Acción genérica.
- Construir edificio.
- Entrenar unidad.
- Construir extractor.
- Recolección total de mineral.
- Recolección total de vespeno.

3.3. Entorno operacional

En este apartado se describe el hardware y software necesarios para el funcionamiento del sistema. El hardware mínimo necesario es:

- Ordenador de sobremesa: este ordenador debe tener por lo menos dos núcleos a 2Mhz, 4 GB de memoria RAM disponible y una tarjeta gráfica con 64 megabytes.

- Router: un router que permita la conexión del sistema.

El software necesario para esta configuración es:

- Sistema operativo base: Linux.

- Java: en su versión 1.7 o superior.

- Metric-FF: es un sistema de planificación independiente de dominio que opera con PDDL. Es una extensión del planificador FF capaz de almacenar un determinado conjunto de variables numéricas en los estados

- VirtualBox: o cualquier solución de emulación que permita la conexión del sistema a internet. El uso de esta herramienta es opcional si se dispone de dos máquinas conectadas con los dos sistemas operativos necesarios.

- Sistema operativo de la máquina virtual: Windows XP.

- PELEA: es una arquitectura capaz de integrar la planificación y la ejecución de los planes generados. Requiere de un planificador, un dominio y un problema.

- StarCraft: es el juego de Blizzard StarCraft con su expansión Broodwar.

- ChaosLauncher: es un proyecto de código abierto consistente en un lanzador para StarCraft que proporciona la habilidad para inyectar *plugins* y ficheros DLL.

- BWAPI: es un proyecto de código abierto que provee de las herramientas necesarias para poder generar una DLL que, inyectada en el juego StarCraft permita controlar el comportamiento de un jugador. BWAPI permite realizar operaciones con unidades, enviar mensajes o consultar datos referentes al mapa tales como consultas sobre el terreno en una determinada posición, las características del mapa y el estado de unidades entre otros.

El diagrama con los elementos mencionados, sus relaciones con el sistema y los ficheros necesarios y producidos se muestran en la figura 12.

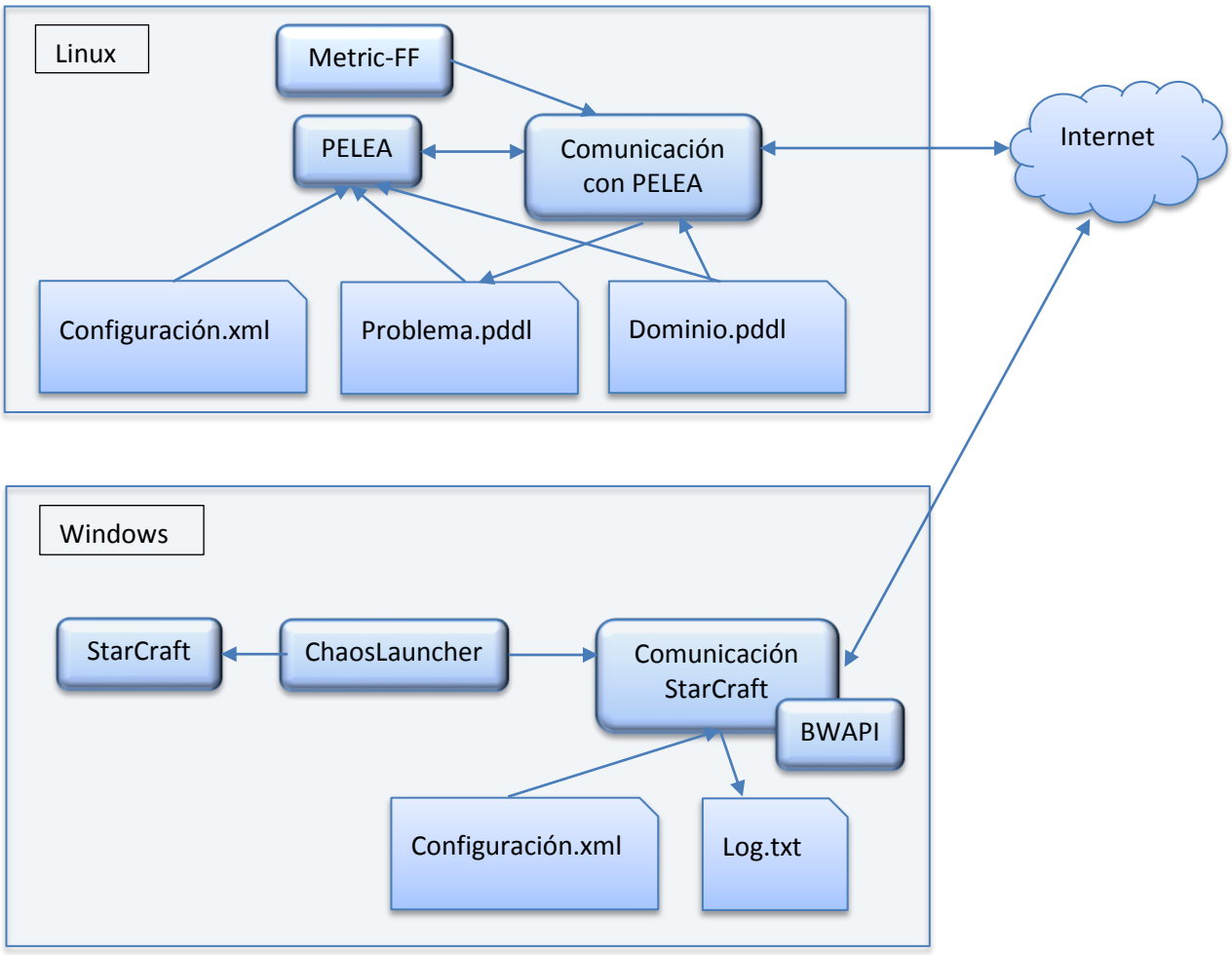

Figura 12: Entorno operacional.

Los módulos del sistema mostrados son:

- Módulo de comunicación con StarCraft: es la aplicación desarrollada en la parte del sistema Windows. Se encarga de almacenar el estado, transformarlo a PDDL y ejecutar las acciones que se requieran entre otros.

- Módulo de comunicación con PELEA: es la aplicación desarrollada en el sistema para Linux. Se encarga de gestionar la comunicación con el módulo de comunicación con StarCraft y de transmitir los mensajes de PELEA y los resultados de los mensajes recibidos a la misma. Permite realizar una planificación limitada.

Los ficheros que se requieren para el funcionamiento son:

- Configuración: contiene la información para el funcionamiento de PELEA y del módulo de comunicación con StarCraft.

- Log: es un fichero que crea el módulo de comunicación con StarCraft registrando mensajes del sistema.

- Problema: es la representación del estado y las metas del juego en un fichero PDDL.

- Dominio: es el fichero de definición del dominio en PDDL.

3.4. Descripción detallada

En este apartado se va a especificar de una forma más precisa el sistema implementado desde su arquitectura general a sus componentes. Se describen también las acciones que se han diseñado, su aplicación en un dominio generado y su implementación. Se concluye el apartado con diagramas que contienen el flujo general de ejecución del sistema.

3.4.1. Diseño arquitectónico

En este apartado se describe el diseño arquitectónico del sistema. Este diseño se encuentra modularizado para facilitar su reutilización y la inclusión de funcionalidades posteriores. Para ello, la arquitectura del sistema es cliente - servidor que se ilustra en la figura 13.

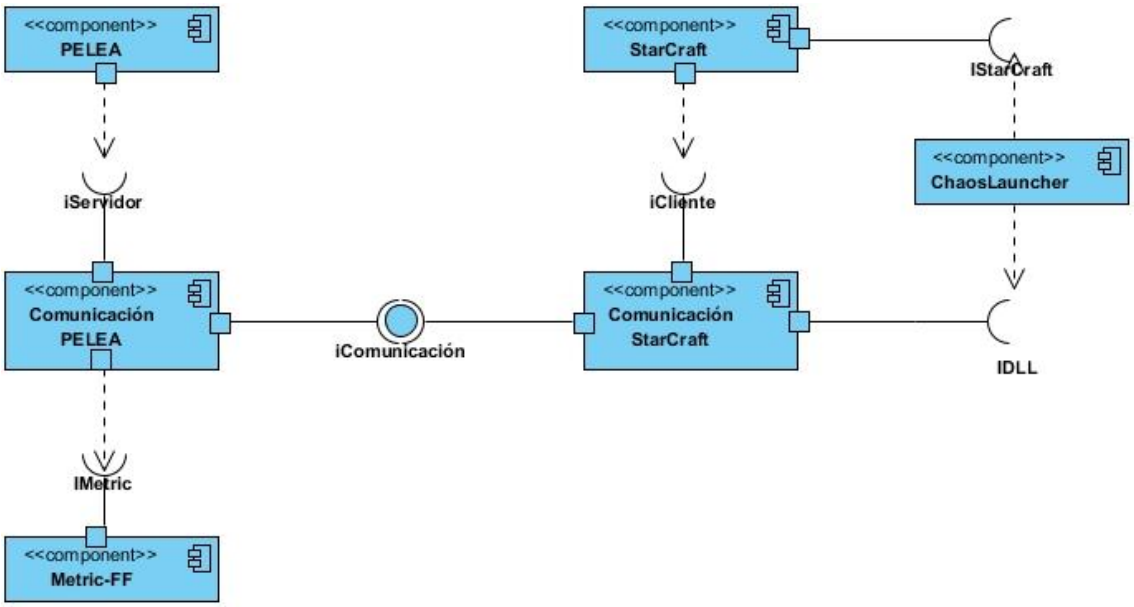

Figura 13: Arquitectura del sistema

Los nodos mostrados son:

- PELEA: arquitectura reducida de PELEA encargada de la planificación, el control y la ejecución.

- StarCraft: es el objetivo de aplicación del sistema y se encuentra instalado en el sistema operativo Windows. No ofrece un API para interactuar directamente, por lo que es necesario el uso de un API generado por terceros, BWAPI y un lanzador que permita la inyección de la misma, ChaosLauncher. Es, por tanto, el juego de estrategia objetivo de la inyección de la DLL generada sobre el módulo de Comunicación StarCraft y que

realizará las llamadas a los eventos que se produzcan en el mismo además de recibir las órdenes de actuación que dicho módulo a través de BWAPI le ordene.

- ChaosLauncher: programa encargado de lanzar StarCraft y de realizar la inyección de una DLL que contenga BWAPI para poder interactuar con dicho juego y recibir los eventos que en él ocurran.

- Comunicación PELEA: el subsistema Comunicación PELEA actúa como servidor y se encarga de gestionar la comunicación entre PELEA y Comunicación StarCraft y de interpretar los comandos del usuario en el servidor. Este subsistema se ha desarrollado para ser completamente independiente del subsistema comunicación StarCraft excepto para el formato de los mensajes que son intercambiados. Esta aproximación permite la reutilización del código de ambas partes para proyectos futuros. Así, el subsistema de comunicación con PELEA actúa de apoyo para efectuar la planificación a un subsistema cliente que será el encargado de ejecutar y controlar la ejecución de las acciones que el subsistema de comunicación con PELEA le indique efectuar. Además, este componente es capaz de ejecutar los comandos del usuario en la consola que provee el mismo, bien mediante la proporcionada al ejecutar el módulo *Execution* de PELEA modificado o bien en la consola al lanzar un servidor básico desligado de PELEA.

- Comunicación StarCraft Este subsistema actúa como cliente y es el encargado de comunicar las acciones con StarCraft así como de controlar su ejecución, gestionar los mensajes con el servidor, mantener el estado interno y transformar el estado a PDDL.

- Metric-FF: para realizar la planificación, el subsistema de comunicación con PELEA deberá contener un fichero PDDL con el dominio en el cuál realizar la planificación y un planificador. En este proyecto se ha hecho uso del planificador Metric-FF para este fin tanto desde PELEA cómo desde el módulo de Comunicación PELEA.

Tanto el componente de comunicación con StarCraft como el de comunicación con PELEA permiten la ejecución de comandos públicos y privados. Esta diferenciación de comandos permite la creación y ejecución de funcionalidades dependiendo del emisor del mensaje. Si el emisor es local, es decir, directamente sobre el componente, estos mensajes son primero tratados y evaluados como privados y posteriormente como públicos. Los objetivos de esta distinción son:

- En primer lugar, ofrecer la posibilidad de ejecución de órdenes dirigidas a cada parte por separado, de esta forma es posible indicar a cada parte que se requiere del sistema.

- En segundo lugar, ofrecer la posibilidad de definición de comandos privados, que sólo un usuario que interactúe directamente con esa parte pueda ejecutar. Así, se permite reescribir un comportamiento implementado como público con nuevas opciones para el usuario privado y la definición de nuevos comportamientos que no es necesario comprobar en público.

3.4.1.1. *PELEA*

En el uso de PELEA, no se ha utilizado su versión completa, sino una reducida. En dicha versión se utilizarán tres de los módulos ya programados, quedando la arquitectura reducida de PELEA como se muestra en la figura 14:

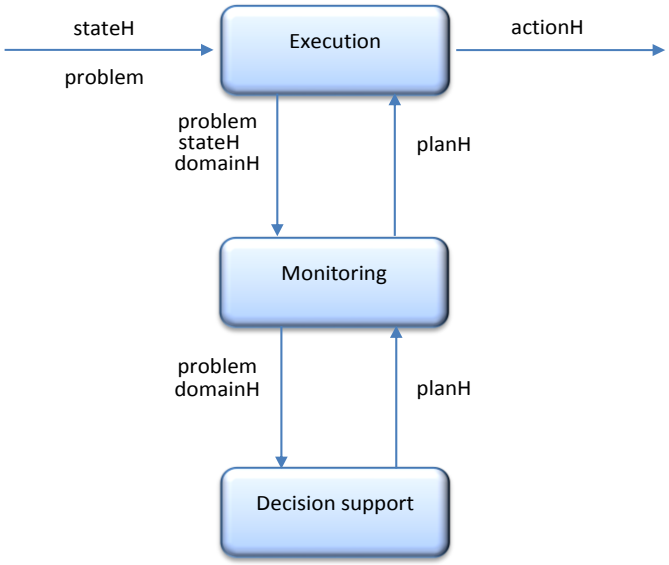

Figura 14: Arquitectura reducida de PELEA

Donde:

- *state* representa un estado con la información de los sensores.
- *domain* representa el dominio del problema
- *problem* indica el problema a resolver para el estado actual
- *H* Indica alto nivel. Un nivel bajo indica acciones simples mientras que un nivel alto, acciones complejas o acciones compuestas de varias acciones simples o complejas.
- *action* representa una acción ejecutable.

En este proyecto se han efectuado cambios sobre la clase del módulo *Execution* que recibe la misma denominación y que es la encargada de realizar las tareas del mismo para que sea capaz de hacer uso del módulo de comunicación PELEA, redirigiendo los mensajes que *Execution* ordenaría a los sensores al módulo de comunicación para su transmisión hacia el módulo ce Comunicación con StarCraft. Además se ha incluido el programa Metric-FF para permitir que PELEA lleve a cabo su tarea de planificación.

3.4.1.2. *Comunicación PELEA*

Este módulo no se descompone en ninguno de nivel inferior y contiene las tres clases que se muestran en la figura 15.

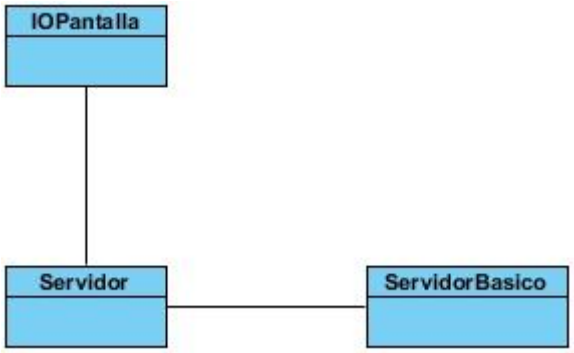

Figura 15: Componente comunicación PELEA

Los elementos mostrados son:

- IOPantalla: clase de apoyo que gestiona la entrada por pantalla de forma paralela en el subsistema de comunicación con PELEA. Contiene la lista de comandos privados.

- Servidor: clase principal del componente, se encarga de la comunicación con el cliente, de ser capaz de aceptar conexiones, tratar y decodificar los mensajes y de ejecutar los comandos públicos. También se encarga de transformar el mensaje del estado recibido en PDDL mediante su guardado a un fichero de forma independiente de la plataforma y de realizar una planificación básica haciendo uso del programa Metric-FF. El envío y recepción de mensajes se efectúa de forma paralela.

- ServidorBásico: esta clase es la encargada de crear un servidor capaz de realizar las comunicaciones con el cliente, ofrecer una funcionalidad básica de planificación y de lanzar el componente sin PELEA.

3.4.1.3. *Comunicación StarCraft*

Dada la complejidad del subsistema de comunicación con StarCraft, se ha utilizado otra arquitectura para su funcionamiento, modelo – vista – controlador (MVC) como se muestra en la figura 16.

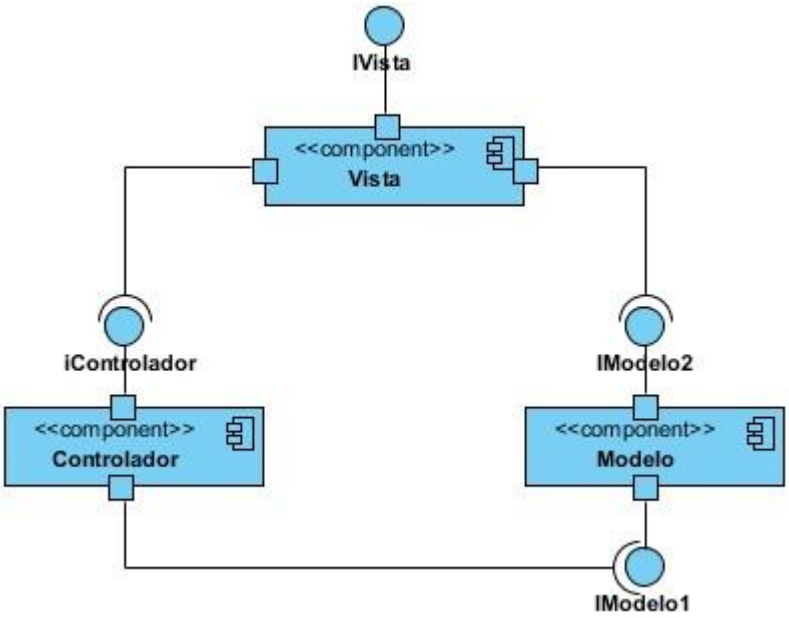

Figura 16: Diagrama de arquitectura del cliente

En esta arquitectura, el componente **vista** será el encargado de realizar las funciones de percibir y actuar del *bot* así como comunicarse con el servidor, **modelo** almacenará la representación interna del estado y **controlador** será el encargado de responder a los eventos producidos en el juego.

Vista

El componente vista se encuentra formado por los componentes que se muestran en la figura 17.

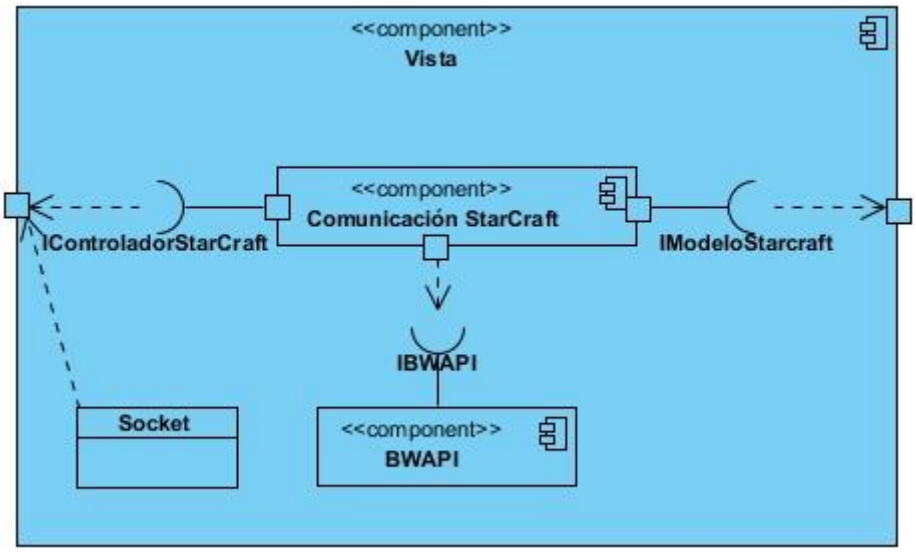

Figura 17: Diagrama de componentes de vista

Los elementos mostrados son:

- Comunicación StarCraft: Vista se encuentra compuesto únicamente de otro subcomponente, Comunicación StarCraft, encargado de gestionar las comunicaciones con el juego, a través de los actuadores, sensores y el código necesario para generar un fichero DLL que sea posible inyectar mediante ChaosLauncher.

- Socket: es la clase encargada de realizar el envío y recepción de mensajes con el Servidor, almacenándose estos en un buzón de mensajes tras realizar una sincronización del hilo que se crea de la misma clase y que será encargado de realizar una escucha permanente en el puerto de comunicación. Esta clase, por tanto, detecta entre los datos recibidos por el *socket* de Windows el principio y el fin de un mensaje del servidor y lo almacena individualmente en el buzón realizando para ello la sincronización entre los hilos de la aplicación y escucha.

Subcomponente comunicación StarCraft

Dentro del componente vista se encuentra el subcomponente Comunicación StarCraft, que en su base define dos clases: “DLL” y “Sensores” e incluye otro subcomponente, Actuadores tal y como se muestra en la figura 18.

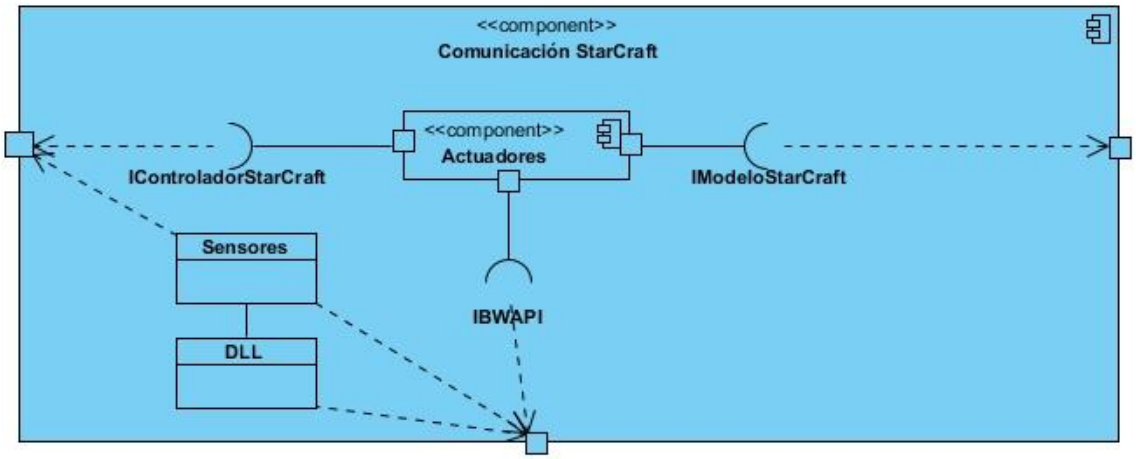

Figura 18: Subcomponente Comunicación StarCraft

Los elementos mostrados son:

- DLL: es una clase que contiene el código necesario por el entorno de desarrollo para generar un fichero DLL capaz de ser inyectado por ChaosLauncher. Este archivo es generado automáticamente al compilar el proyecto desde el entorno de desarrollo.

- Sensores: es la clase encargada de recibir los eventos del juego así como almacenar una instancia del mismo. Además, es la encargada de crear un objeto de la clase "Nucleo" del controlador e inicializar los valores de la misma. También realiza la detección inicial de los recursos cercanos y unidades del jugador así como inicializar en el estado las funciones iniciales. Es la encargada además de actualizar los valores de las funciones que son accesibles directamente desde el juego como el desarrollador considere oportuno. Por último, identifica y ejecuta los comandos privados que el usuario del cliente envíe a través de la consola del juego.

- Actuadores: dentro del subcomponente Comunicación con StarCraft se incluye otro subcomponente, Actuadores, que englobará todos los actuadores definidos por el desarrollador del *bot*.

Actuadores

En este componente se encuentran las clases que definen los distintos actuadores implementados en el sistema y que hacen uso desde la clase Actuador de todas las interfaces mostradas en el subcomponente Comunicación StarCraft. Dichas clases se recogen en la figura 19.

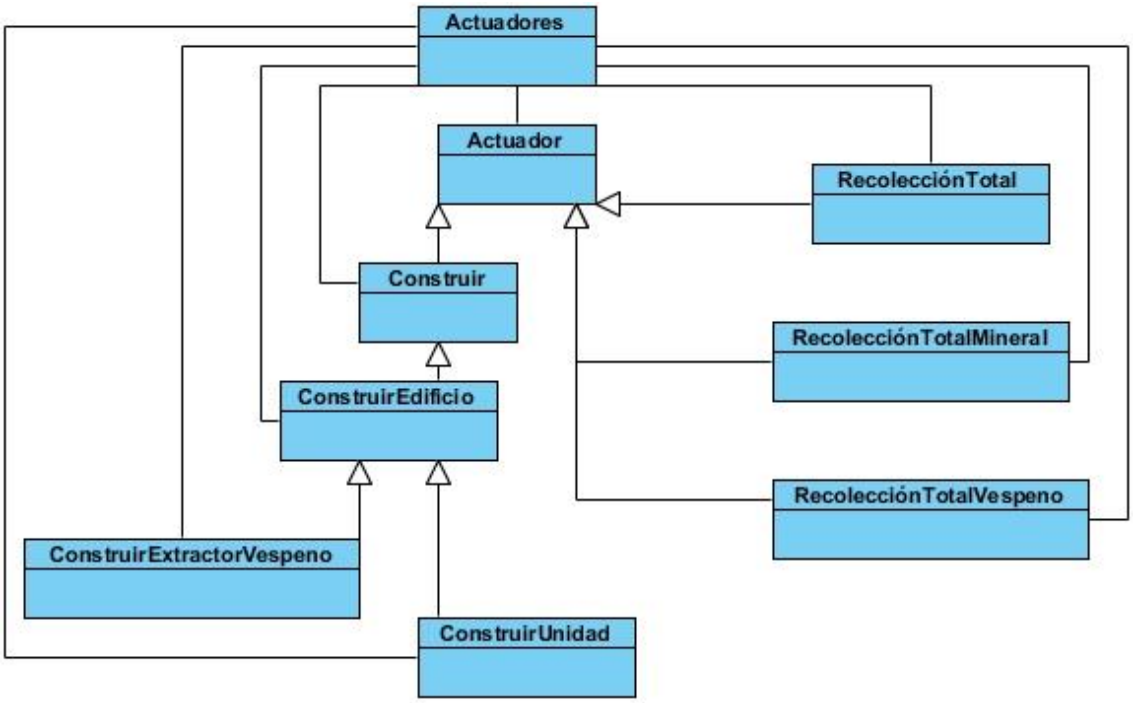

Figura 19: Subcomponente actuadores

Estas clases son:

- Actuadores: esta clase es el nexo principal de unión del componente con el controlador y contiene la implementación de las acciones programadas mediante retro-llamada. Se encarga de almacenarlas en el dominio y de crear un nuevo actuador e inicializarlo con la información procedente del servidor.

- Actuador: de esta clase heredan todas las acciones definidas por el usuario mediante orientación a objetos. Esta clase se define en mayor detalle puesto que en ella se almacenan los efectos tanto en funciones como en predicados que la acción efectúa al cumplir correctamente su ejecución además de:

  - Una representación interna en forma de cadena de caracteres del estado del actuador y una cantidad de tiempo en *frames* que dicho actuador tiene como espera.

  - Si la acción se puede cancelar y si ésta se encuentra en ejecución

  - El nombre y la definición de los parámetros que recibe de entrada tal y como sean indicados por el mensaje de ejecución del servidor.

  - Los objetos del estado del juego en función de los parámetros del actuador en forma de puntero genérico *void*

  - Un enlace para obtener las definiciones y objetos contenidos en el modelo así como para realizar los cambios.

  - Un enlace a la instancia del juego donde el actuador efectúa los comportamientos.

- Las funciones que provee son las relativas a la gestión de los datos anteriores y su consulta además de la automatización de la obtención de los objetos del actuador en función de los parámetros que le sean indicados. Incluye también la funcionalidad para hacer efectivos automáticamente los efectos almacenados en el actuador sobre el estado que tenga enlazado y para imprimir mensajes en el juego guardando un registro inmediato de los mismos. Finalmente, funciones que deberán ser sobrescritas por los actuadores que el desarrollador cree para poder incluir los comportamientos que se deseen así como para incluir los cambios parciales o totales que su ejecución produce en el estado y el control del tiempo de espera.

- RecolecciónTotal: es el primer actuador implementado y que contiene una comprobación sobre la extracción de recursos.

- RecolecciónTotalMineral: contiene el comportamiento de una acción que ordena a los trabajadores del juego recolectar mineral.

- RecolecciónTotalVespeno: homóloga a la anterior, permite la extracción de vespeno.

- Construir: esta clase abstracta contiene la definición de una máquina de estados capaz de dirigir la construcción en el juego.

- ConstruirEdificio: es una extensión de la clase anterior que implementa los *scripts* necesarios para la construcción de un edificio.

- ConstruirExtractorVespeno: una especialización de la clase anterior que permite construir el edificio especial para la extracción de vespeno.

- ConstruirUnidad: esta clase hereda su comportamiento de Construir y permite el entrenamiento de nuevas unidades.

Modelo

Este componente está compuesto por dos subcomponentes tal y como se detalla en la figura 20.

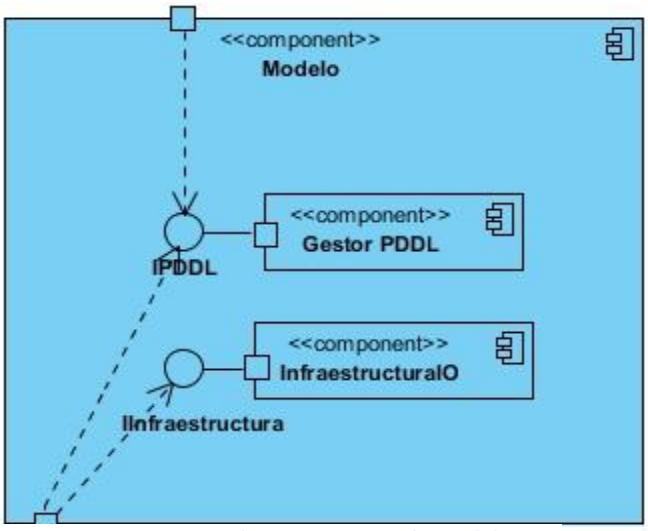

Figura 20: Diagrama de componentes de modelo

Este componente se encuentra dividido en dos subcomponentes:

- Gestor PDDL: este elemento almacena una descripción del estado del juego en dos formatos:

  - Una representación de la información del entorno mediante objetos que son almacenados en estructuras con respecto a sus características.

  - Una representación del estado del juego en formato PDDL mediante una representación basada en predicados, funciones y acciones.

    Además, este componente ofrece un conjunto de métodos para realizar transformaciones entre los dos formatos y permitir transformar el estado del juego a PDDL.

- InfraestrucutraIO: permite el uso de los ficheros que utiliza el sistema, tanto de configuración como de registro. Incluye además las representaciones de objetos definidos por el desarrollador de *bots* en forma de estructuras.

Gestor PDDL

Este componente provee de las clases mostradas en la figura 21. La funcionalidad de las interfaces que este componente proporciona al resto del sistema se encuentra dentro de la clase "Dominio" y cada una de las clases del componente PDDL tiene definida una función que transforma lo que representa a PDDL.

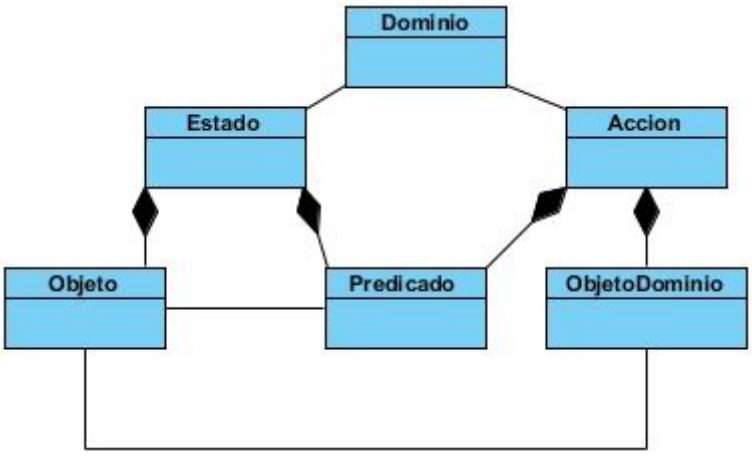

Figura 21: Componente gestor PDDL

- Dominio: El componente PDDL presenta la clase Dominio, encargada de ofrecer una interfaz para actuar con el resto de componentes del mismo además de almacenar los punteros a las acciones incluidas por retro-llamada y que son almacenadas mediante instancias de la clase "Accion". La clase Dominio se encarga, además, de realizar la llamada de ejecución de la acción, enlazando las instancias de los objetos necesarios por parámetros a la retro-llamada. Posteriormente, se hacen efectivos los cambios de la acción correspondientes a los predicados definidos en la misma enlazando automáticamente los punteros de los objetos que la acción recibe por parámetros. Finalmente, en esta clase se definen de manera estática los objetivos para el estado del juego.

- Estado: representa y gestiona una instancia concreta en la cual se almacenan los objetos del mismo, así como los predicados y funciones que lo definen. Proporciona además la funcionalidad para entre dichos objetos en función del nombre que se le haya dado para su representación en PDDL. Estos objetos son almacenados mediante las clases Objeto y ObjetoDominio.

- Accion: almacena el puntero de retro-llamada y guarda una descripción de la acción con su nombre, sus parámetros y los cambios que efectúa la ejecución de la misma en el estado.

- Objeto: almacena el nombre que se le ha dado al objeto en el estado, el tipo del mismo y un puntero al objeto almacenado.

- Predicado: almacena una representación del valor que los predicados toman en el estado. Para ello almacena el nombre y una lista con punteros a las instancias de Objeto que representan los parámetros del mismo. Además, almacena un puntero a una retro-llamada que se ejecutará incluyendo por parámetro el estado, alta o baja del mismo. Esta misma clase es utilizada para la representación de funciones

- ObjetoDominio: la segunda representación de objetos se corresponde con la del dominio. Esta representación se encuentra contenida en ObjetoDominio, que almacena el tipo del objeto y el pseudonombre que se le da al mismo en el dominio y, en caso de ser instanciado para una acción, un puntero a una instancia de la clase Objeto que represente el objeto en el estado

InfraestructuraIO

El segundo de los componentes, infraestructura, contiene las clases utilizadas como apoyo por el resto de la aplicación, tanto para los objetos definidos por el usuario como para la entrada y salida a ficheros. Las clases se muestran en la figura 22.

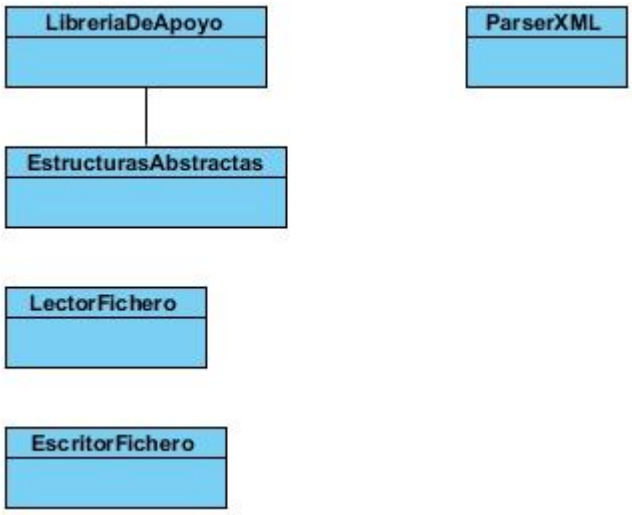

Figura 22: Componente InfraestructuraIO

Las clases mostradas se corresponden con:

- LibreriaDeApoyo: contiene la definición de unas funciones de uso común por muchas de las clases del sistema como son la conversión de cadena de caracteres a texto, añadir un número a una cadena de texto o la comparación de dos cadenas de texto ignorando mayúsculas o minúsculas.

- EstructurasAbstractas: contiene la definición de todas las estructuras de objetos que el desarrollador de un *bot* defina. En ella se definirán también las actualizaciones bajo demanda de las funciones relativas a dichas estructuras. Se encuentra definida la estructura "base" y la función que recibe una base y un dominio y actualiza los valores requeridos de la misma en dicho dominio. De esta forma, se consigue el mismo efecto que recoger el valor de las funciones directamente del juego en los sensores cuando el envío del estado es requerido por el servidor. Facilita además el control de dichos valores, ya que no será necesario actualizar el valor cuando por ejemplo, una unidad ha sido destruida consiguiendo de esta forma tres efectos beneficiosos:

  - La actualización de funciones sobre los objetos creados se ve asegurada frente a errores indirectos, tales como las unidades en cola de construcción de un edificio destruido. Este efecto se produce al estar centralizado el control de dichos valores dentro de esta clase, no siendo necesario incluir las actualizaciones de forma programada desde los sensores o actuadores.

  - Se obtiene mejor rendimiento al no estar dichos valores en continua actualización en el estado del juego además de en el objeto que representa la información.

  - Los valores de funciones que almacenen cantidades de unidades se pueden actualizar de forma trivial: Dichas unidades se pueden almacenar como conjuntos, de tal forma que su localización no incluya una consulta mediante los sensores de todas las unidades del jugador que cumplan los requisitos que defina la función. De esta forma, se facilita la localización de las unidades a los actuadores, ya que quedan almacenadas en el estado en la representación que el desarrollador defina, sin tener que realizar operaciones complejas sobre los sensores para calcular dichos conjuntos en el momento de la ejecución de un actuador.

- Lector: provee de la funcionalidad para leer todo el contenido de un fichero a una lista de cadenas de texto. Dicha lista almacena los datos correspondientes a una línea de texto del fichero que se encuentre separada por un retorno de carro.

- Escritor: provee dos funciones, para el guardado del registro de sucesos del juego y para anexionar una cadena de texto a un fichero cuya ruta se le indique de forma atómica.

- ParserXML: permite la creación de una cadena de texto que contenga un XML con un estado PDDL limitado, siendo el contenido del estado indicado manualmente a través de sus funciones para añadir un objeto, un objetivo y un contenido a la etiqueta init de PDDL.

Controlador

El componente controlador se compone únicamente de la clase núcleo que se encarga de responder a los eventos producidos en el sistema tanto en el juego como por mensajes recibidos del servidor además de:

- La ejecución de los comandos públicos del cliente.
- Los modos de ejecución de las acciones, paralela o secuencial.
- Enlace de los mensajes con acciones del servidor.
- Control de la ejecución de las acciones.
- Gestión de las respuestas de las ejecuciones de las acciones.
- Tratamiento de los mensajes del servidor.
- Almacenar las instancias del dominio, actuadores, escritor, estructuras abstractas y configuración del cliente.
- Nombrar los objetos del estado de forma automática así como almacenar los tipos de los mismos.
- Almacenar los objetos representativos de las bases.

Diagrama de clases

Para la mejor comprensión del componente se expone un diagrama de clases general que muestra las asociaciones a nivel global del mismo en la figura 23.

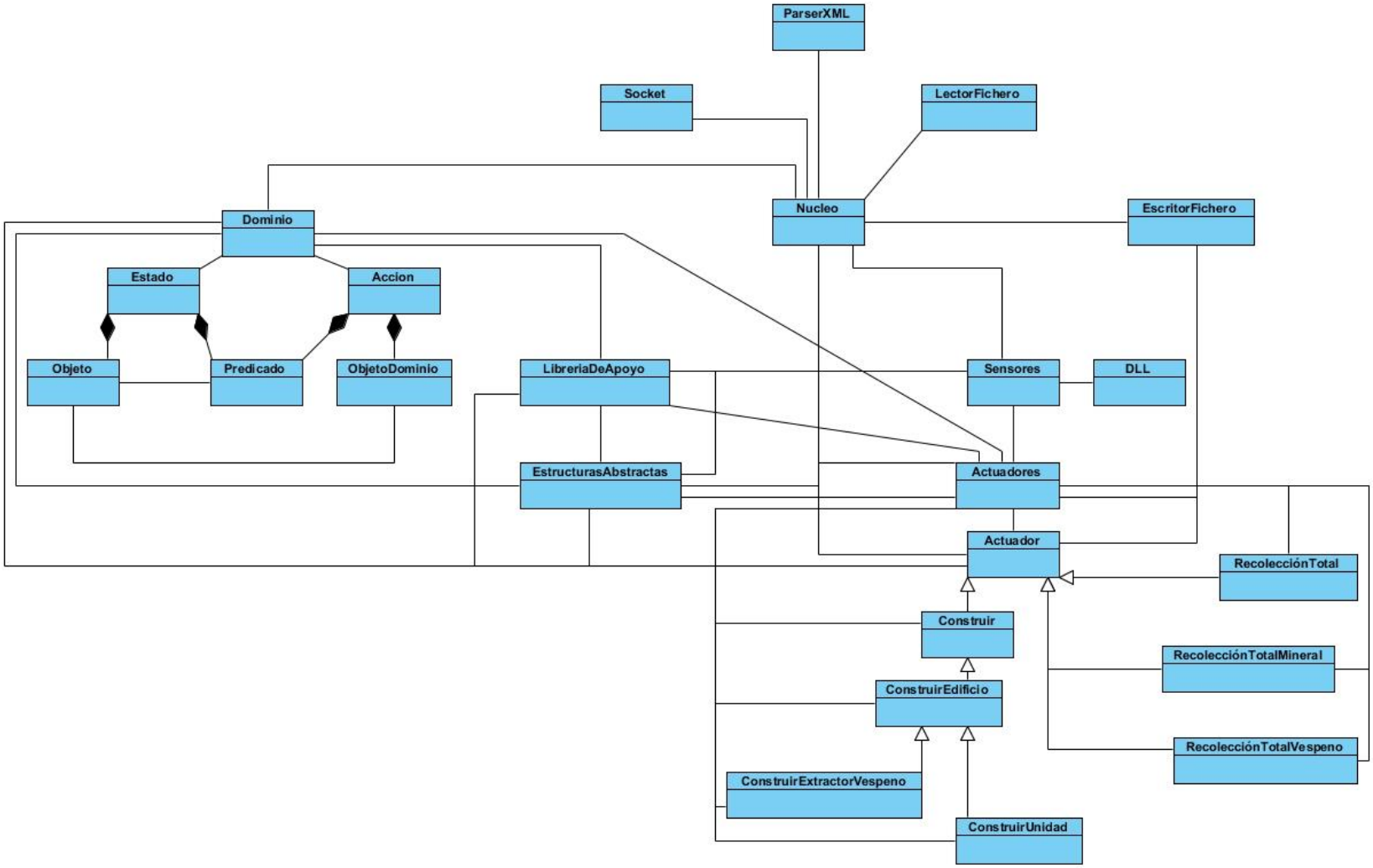

Figura 23: Diagrama de clases del componente comunicación StarCraft

3.4.2. Representación de la información

En esta apartado se realiza una descripción detallada de los diferentes modelos de representación que han sido utilizados para definir la información del entorno. Como se describió anteriormente se han realizado dos representación diferentes, una más cercana a la forma que utiliza el juego para representar la información basada en objetos (bajo nivel) y otra basada en lógica de predicados de forma que pueda ser utilizada para aplicar planificación automática como modelo de razonamiento (alto nivel).

3.4.2.1. *Conocimiento de alto nivel*

Como se ha comentado anteriormente, para poder utilizar la planificación es necesario el uso de dos ficheros, dominio y problema en PDDL que representen la descripción del entorno en el cuál se opera, el instante del problema en el que el juego se encuentra y las metas a alcanzar.

Representación del estado

Para la creación del dominio, es necesaria la creación de un fichero PDDL que lo contenga junto con acciones que estén programadas en el cliente que se deseen utilizar. Sobre este dominio se plasma el estado y los objetivos del mismo del *bot*. Ambos, el estado y los objetivos, se generan desde el cliente en la implementación actual. Las características del dominio ejemplo generado capaz de interactuar con el sistema se describen en las dos tablas mostradas a continuación y que contienen los tipos de objetos, predicados y funciones que se han utilizado.

Tipos de objetos	Significado
Deposito_mineral	Este objeto representa un depósito de mineral del juego por si se desea el control planificado de la asignación de los trabajadores a la recolección.
Terran_SCV	Representa la unidad Terran SCV.
Trabajador	Este objeto es una abstracción de todos las unidades trabajadoras de cada raza que puede usarse para la asignación planificada de recolección, construcción…
Terran_Command_Center	Este objeto representa el centro de mando Terran.
Base	Este abstracto no existente en el juego denota una posición a la cual se vinculan edificios, unidades, depósitos (si se controla el reparto de trabajadores desde el cliente)…
Juego	Representa la instancia del juego sobre el cual el actuador opera.
Geiser_vespeno	Representa los geiseres de vespeno del juego.

Tabla 68: tipos de objetos

Los predicados que el dominio acepta se especifican a continuación:

Predicados	Significado
Poblacion_maxima ?j – Juego	Indica que se ha alcanzado la población máxima disponible para el juego.
Trabajadores_disponibles ?b –Base	Indica que la base dispone de trabajadores cuya categoría dentro de la misma es desocupado.
Recoleccion_total_mineral ?b – Base	Señala que en la base se ha alcanzado el nivel de recolección de minerales máximo.
Recoleccion_total_vespeno ?b – Base	Señala que en la base se ha alcanzado el nivel de recolección de vespeno máximo.
Recoleccion_total ?b – Base	Indica que en la base se ha conseguido el nivel de recolección máxima tanto para vespeno como para minerales.
Extractor_vespeno ?b – Base	Almacena si en la base se dispone de un extractor de vespeno del cual se pueda extraer el gas.
Puede_crear_unidades ?b – base	Indica si una base puede crear unidades. Esto es posible si dicha base dispone al menos de un centro de mando.
Puede_construir ?b – base	Marca si una base dispone de espacio para la construcción de nuevos edificios y si se dispone de trabajadores en la misma para realizar dicha tarea.

Tabla 69: predicados del dominio

En este dominio también se hace uso de la posibilidad de incluir funciones por parte del sistema. Las funciones con las que opera el dominio se listan a continuación:

Funciones	Significado
minerales	Almacena la cantidad de minerales que el jugador posee en reserva.
vespeno	Almacena la cantidad de vespeno que el jugador posee en reserva.
frames	Indica el número de *frames* transcurridos en el juego.
población_actual	Almacena la cantidad de población utilizada.
población_maxima	Indica la población máxima que dispone el jugador.
depositos_mineral	Contabiliza los depósitos de mineral que se encuentran al alcance de la base inicial del jugador.
geiseres_vespeno	Contabiliza los depósitos de vespeno que se encuentran al alcance de la base inicial del jugador.
trabajadores_mineral	Almacena la cantidad de trabajadores que se encuentran extrayendo mineral.
trabajadores_vespeno	Almacena la cantidad de trabajadores que se encuentran extrayendo vespeno.
trabajadores_desocupados	Indica la cantidad de trabajadores dentro de la categoría “desocupados”.
cuarteles	Señala la cantidad de cuarteles construidos.
factorías	Almacena el número de factorías creadas.
puertosespaciales	Indica la cantidad de puertos espaciales construidos.
academias	Señala la cantidad de academias construidas.
ingenierías	Almacena el número de bahías de ingeniería creadas.
armerías	Indica la cantidad de armerías construidas.
marines	Señala la cantidad de marines disponibles.
firebats	Almacena el número de *firebats* entrenados.

Funciones	Significado
buitres	Indica la cantidad de buitres disponibles.
médicos	Señala la cantidad de médicos entrenados.
goliats	Almacena el número de goliats disponibles.
espectros	Indica la cantidad de espectros entrenados.

Tabla 70: funciones del dominio

Finalmente, para el establecimiento de los objetivos, se realizan de manera estática mediante código programado dentro de la clase Dominio. De la misma forma, se definen los requisitos del estado para el planificador y la métrica para minimizar o maximizar en base a funciones del estado. Los objetivos que son añadidos por el *bot* programado en este proyecto en PDDL son los siguientes:

Objetivo	Significado
(Recoleccion_total Base1)	Se requiere que la Base1, inicial, disponga de los suficientes trabajadores extrayendo recursos de tal forma que se maximice la recolección de los mismos.
(>= (marines) 20)	Se requiere de 20 unidades del tipo “marine”
(>= (firebats) 10)	Se requiere de 10 unidades del tipo “*firebat*”
(>= (médicos) 8)	Se requiere de 8 unidades del tipo “médico”
(>= (buitres) 5)	Se requiere de 5 unidades del tipo “buitre”
(>= (goliats) 3)	Se requiere de 3 unidades del tipo “goliats”
(>= (espectros) 2)	Se requiere de 2 unidades del tipo “espectro”
(:metric minimize (frames))	Establece el objetivo del planificador como la minimización de la función *frames*. Siendo los *frames* una medida de tiempo correspondiente al repintado de pantalla del juego.

Tabla 71: objetivos del dominio

Estos objetivos tienen el propósito de permitir la generación de un plan complejo para establecer una base operativa inicial con un ejército capaz de protegerla. Estos objetivos son generados cada vez que se requiere el estado al cliente. Además, dados los requisitos de entrenamiento de las unidades expuestos en el dominio que se muestra a continuación, esta base dispondrá de infraestructuras versátiles para la continuación de la partida.

Un ejemplo inicial del estado que se obtiene para este dominio en la aplicación se corresponde con el representado en la figura 24.

```
(define (problem problema1) (:domain StarCraft)
(:objects
        Base0 - Base
        Juego0 - Juego
        Base1 - Base
)
(:init
        (Trabajadores_disponibles Base1)
        (Puede_crear_unidades Base1)
        (Puede_construir Base1)
        (= (cuarteles) 0)
        (= (factorias) 0)
        (= (puertosespaciales) 0)
        (= (academias) 0)
        (= (ingenierias) 0)
        (= (armerias) 0)
        (= (marines) 0)
        (= (firebats) 0)
        (= (buitres) 0)
        (= (medicos) 0)
        (= (goliats) 0)
        (= (espectros) 0)
        (= (minerales) 50)
        (= (vespeno) 0)
        (= (frames) 4)
        (= (poblacion_actual) 8)
        (= (poblacion_maxima) 20)
        (= (depositos_mineral) 8)
        (= (geyseres_vespeno) 1)
        (= (trabajadores_mineral) 0)
        (= (trabajadores_vespeno) 0)
        (= (trabajadores_desocupados) 4)
)
(:goal
        (and (Recoleccion_total Base1)
             (>= (marines) 20)
             (>= (firebats) 10)
             (>= (medicos) 8)
             (>= (buitres) 5)
             (>= (goliats) 3)
             (>= (espectros) 2))
)
(:metric minimize (frames)))
```

Figura 24: Estado inicial

En dicha figura se puede comprobar que el sistema ha reconocido los siguientes objetos:

- Base0: identifica a la base nula, es decir, la base que no posee ningún edificio. Es la base para las unidades que no han sido asignada a ninguna.

- Juego0: es una referencia del juego que aplica el estado, necesario para algunas acciones dentro del sistema.

- Base1: es el objeto que hace referencia a la base inicial del jugador, con las unidades iniciales asociadas a ella.

Los valores de iniciación de las funciones están todos a cero, puesto que no se dispone de ninguna de las unidades que contabilizan, salvo para *frames* que indica el número transcurrido de estos en la partida, la cantidad de minerales disponibles y las funciones referentes a la población utilizada y trabajadores disponibles.

Los hechos que están activos son los referentes a la posibilidad de creación de unidades, dado que la base dispone de un edificio central capaz de entrenar trabajadores, el indicativo de que existen trabajadores disponibles para tareas y que se puede construir debido a que hay trabajadores asociados a la base.

Representación de la acciones

Con los datos definidos anteriormente, el dominio define las siguientes acciones:

- Crear trabajadores: esta acción crea un trabajador si se cumplen los requisitos de recursos. Como efectos consume los recursos necesarios más un tiempo de producción y almacena el trabajador dentro de los desocupados. Su sintaxis se muestra en la figura 25.

```
(:action CrearTrabajadores

        :parameters (?b - Base ?j -Juego)

        :precondition (and
                                (not (Poblacion_maxima ?j))
                                (>= (minerales) 50)
                                (Puede_crear_unidades ?b)
                                (< (poblacion_actual) (poblacion_maxima)))

        :effect (and (Trabajadores_disponibles ?b)
                        (increase (frames) 300)
                        (increase (trabajadores_desocupados) 1)
                        (decrease (minerales) 50)
                        (increase (poblacion_actual) 2)))
```

Figura 25: Crear trabajadores

- RecoleccionTotal: si se ha alcanzado la recolección total de vespeno y minerales, la base se encuentra recolectando al máximo. Su sintaxis se muestra en la figura 26.

```
(:action RecoleccionTotal

        :parameters (?b - Base)

        :precondition (and (Recoleccion_total_mineral ?b)
                           (Recoleccion_total_vespeno ?b))

        :effect (and (Recoleccion_total ?b)))
```

Figura 26: Acción RecoleccionTotal

- RecoleccionTotalMineral: si en una base se consigue que la cantidad de trabajadores que extraen minerales sea dos veces el número de depósitos de mineral, entonces se está extrayendo al máximo de los depósitos de mineral. Su sintaxis se muestra en la figura 27.

```
(:action RecoleccionTotalMineral

        :parameters (?b - Base)

        :precondition (and
                    (>= (trabajadores_mineral) (* (depositos_mineral) 2)))

        :effect (and
                    (Recoleccion_total_mineral ?b)))
```

Figura 27: Acción RecoleccionTotalMineral

- RecolectarMineral: ordena a los trabajadores desocupados de una base que recolecten mineral. Su sintaxis se muestra en la figura 28.

```
(:action RecolectarMineral

        :parameters (?b - Base)

        :precondition (and (Trabajadores_disponibles ?b)
                        (not  (Recoleccion_total_mineral ?b))
                         (> (trabajadores_desocupados) 0))

        :effect (and (not (Trabajadores_disponibles ?b))
                (increase (trabajadores_mineral) (trabajadores_desocupados))
                (assign (trabajadores_desocupados) 0)))
```

Figura 28: Acción RecolectarMineral

- RecoleccionTotalVespeno: si en una base la cantidad de trabajadores que extraen vespeno es tres veces el número de géiseres, entonces se está extrayendo al máximo de los extractores de vespeno. Su sintaxis se muestra en la figura 29.

```
(:action RecoleccionTotalVespeno

        :parameters (?b - Base)

        :precondition (and (>= (trabajadores_vespeno) (* (geyseres_vespeno) 3)))

        :effect (and (Recoleccion_total_vespeno ?b)))
```

Figura 29: Acción RecoleccionTotalVespeno

- RecolectarVespeno: ordena a los trabajadores desocupados de una base que recolecten vespeno. Su sintaxis se muestra en la figura 30.

```
(:action RecolectarVespeno

      :parameters (?b - Base)

      :precondition (and (Trabajadores_disponibles ?b)
                         (Extractor_vespeno ?b)
                         (not  (Recoleccion_total_vespeno ?b))
                         (> (trabajadores_desocupados) 0))

      :effect (and
                  (not (Trabajadores_disponibles ?b) )
                  (increase (trabajadores_vespeno) (trabajadores_desocupados))
                  (assign (trabajadores_desocupados) 0)))
```

Figura 30: RecolectarVespeno

- ConstruirExtractorVespeno: si se puede construir, ordena la construcción de un extractor de vespeno en una base. Su sintaxis se muestra en la figura 31.

```
(:action ConstruirExtractorVespeno

        :parameters (?b - Base)

        :precondition (and (not (Extractor_vespeno ?b))
                           (>= (minerales) 100)
                           (Puede_construir ?b))

:effect (and (Extractor_vespeno ?b)
             (decrease (minerales) 100)
             (increase (frames) 300)))
```

Figura 30: Acción ConstruirExtractorVespeno

- ConstruirDeposito: si se puede construir, ordena la construcción de un depósito de vespeno en una base. Su sintaxis se muestra en la figura 32.

```
(:action ConstruirDeposito

    :parameters (?b - base ?j - Juego)

    :precondition (and  (>= (minerales) 100)
                        (Puede_construir ?b))

    :effect (and (not (Poblacion_maxima ?j) )
                 (decrease (minerales) 100)
                 (increase (frames) 300)
                 (increase (poblacion_maxima) 16)
                 (assign (poblacion_libre) (- (poblacion_maxima) (poblacion_actual))))
```

Figura 32: Acción ConstruirDeposito

- Esperar: esta acción es necesaria para permitir la generación de recursos al planificador basada en la cantidad de trabajadores recolectando con un coste de tiempo. Su sintaxis se muestra en la figura 33.

```
(:action Esperar

    :parameters (?b - base)

    :precondition (> (frames) 0)

    :effect (and (increase (minerales) (* (trabajadores_mineral) 8))
                 (increase (vespeno) (* (trabajadores_vespeno) 8))
                 (increase (frames) 150)))
```

Figura 33: Acción Esperar

- ConstruirCuarteles: si se puede construir ordena la construcción de unos cuarteles en una base. Su sintaxis se muestra en la figura 34.

```
(:action ConstruirCuarteles

        :parameters (?b - Base)

        :precondition (and (Puede_construir ?b)
                           (>= (minerales) 150))

        :effect (and (increase (cuarteles) 1)
                     (decrease (minerales) 150)
                     (increase (frames) 300)))
```

Figura 34: Acción ConstruirCuarteles

- ConstruirAcademia: si se dispone de cuarteles construidos y los recursos, se puede construir una academia. Su sintaxis se muestra en la figura 35.

```
(:action ConstruirAcademia

          :parameters (?b - Base)

          :precondition (and (Puede_construir ?b)
                             (>= (minerales) 150)
                             (>= (cuarteles) 1))

          :effect (and (increase (academias) 1)
                       (decrease (minerales) 150)
                       (increase (frames) 300)))
```

Figura 35: Acción ConstruirAcademia

- ConstruirFactoría: si se han construido los cuarteles, es posible la construcción de una fábrica si se disponen de los materiales para ello. Su sintaxis se muestra en la figura 36.

```
(:action ConstruirFactoria

        :parameters (?b - Base)

        :precondition (and (Puede_construir ?b)
                           (>= (minerales) 200)
                           (>= (vespeno) 100)
                           (>= (cuarteles) 1))

        :effect (and (increase (factorias) 1)
                     (decrease (minerales) 200)
                     (decrease (vespeno) 100)
                     (increase (frames) 300)))
```

Figura 36: Acción ConstruirFactoría

- ConstruirIngeniería: si se puede construir y se disponen de los materiales, se puede construir una bahía de ingeniería. Su sintaxis se muestra en la figura 37.

```
        (:action ConstruirIngenieria

        :parameters (?b - Base)

        :precondition (and (Puede_construir ?b)
                           (>= (minerales) 125))

        :effect (and (increase (ingenierias) 1)
                     (decrease (minerales) 125)
                     (increase (frames) 300)))
```

Figura 37: Acción ConstruirIngeniería

- ConstruirArmería: si se dispone de una fábrica, los recursos necesarios y se puede construir, es posible crear una armería. Su sintaxis se muestra en la figura 38.

```
(:action ConstruirArmeria

:parameters (?b - Base)

:precondition (and (Puede_construir ?b)
                   (>= (minerales) 150)
                   (>= (vespeno) 50)
                   (>= (factorias) 1))

:effect (and (increase (armerias) 1)
             (decrease (minerales) 150)
             (decrease (vespeno) 50)
             (increase (frames) 300)))
```

Figura 38: Acción ConstruirArmería

- ConstruirPuertoEspacial: si se dispone de una fábrica, los recursos necesarios y se puede construir, es posible crear un puerto espacial. Su sintaxis se muestra en la figura 39.

```
(:action ConstruirPuertoEspacial

:parameters (?b - Base)

:precondition (and (Puede_construir ?b)
                   (>= (minerales) 150)
                   (>= (vespeno) 100)
                   (>= (factorias) 1))

:effect (and (increase (puertosespaciales) 1)
             (decrease (minerales) 150)
             (decrease (vespeno) 100)
             (increase (frames) 300)))
```

Figura 39: Acción ConstruirPuertoEspacial

- CrearMarine: si se dispone de espacio de población, se pueden crear unidades, se dispone del resto de recursos y de cuarteles, se puede ordenar el entrenamiento de un marine. Su sintaxis se muestra en la figura 40.

```
(:action CrearMarine

:parameters (?b - Base ?j - Juego)

:precondition (and (not (Poblacion_maxima ?j))
                   (Puede_crear_unidades ?b)
                   (>= (minerales) 50)
                   (<= (+ (poblacion_actual) 2) (poblacion_maxima))
                   (>= (cuarteles) 1))

:effect (and (increase (frames) 300)
             (increase (marines) 1)
             (decrease (minerales) 50)
             (increase (poblacion_actual) 2)))
```

Figura 40: acción CrearMarine

- CrearFirebat: si se dispone de los recursos y una academia, se puede crear un murciélago en los cuarteles. Su sintaxis se muestra en la figura 41.

```
(:action CrearFirebat

:parameters (?b - Base ?j - Juego)

:precondition (and (not (Poblacion_maxima ?j))
                   (Puede_crear_unidades ?b)
                   (>= (minerales) 50)
                   (>= (vespeno) 25)
                   (>= (academias) 1)
                   (>= (cuarteles) 1)
                   (< (+ (poblacion_actual) 2) (poblacion_maxima)))

:effect (and (increase (frames) 300)
             (increase (firebats) 1)
             (decrease (minerales) 50)
             (decrease (vespeno) 25)
             (increase (poblacion_actual) 2)))
```

Figura 41: Acción CrearFirebat

- CrearBuitre: si se dispone de una fábrica se puede ordenar la creación de un buitre. Su sintaxis se muestra en la figura 42.

```
(:action CrearBuitre

:parameters (?b - Base ?j - Juego)

:precondition (and (not (Poblacion_maxima ?j))
                   (Puede_crear_unidades ?b)
                   (>= (minerales) 75)
                   (>= (factorias) 1)
                   (<= (+ (poblacion_actual) 4) (poblacion_maxima)))

:effect (and (increase (frames) 300)
             (increase (buitres) 1)
             (decrease (minerales) 75)
             (increase (poblacion_actual) 4)))
```

Figura 42: Acción CrearBuitre

- CrearMedico: si se dispone de los recursos, un cuartel y una academia, es posible entrenar un médico. Su sintaxis se muestra en la figura 43.

```
(:action CrearMedico

:parameters (?b - Base ?j - Juego)

:precondition (and (not (Poblacion_maxima ?j))
                   (Puede_crear_unidades ?b)
                   (>= (minerales) 50)
                   (>= (vespeno) 25)
                   (>= (academias) 1)
                   (>= (cuarteles) 1)
                   (<= (+ (poblacion_actual) 2) (poblacion_maxima)))

:effect (and (increase (frames) 300)
             (increase (medicos) 1)
             (decrease (minerales) 50)
             (decrease (vespeno) 25)
             (increase (poblacion_actual) 2)))
```

Figura 43: Acción CrearMedico

- CrearGoliat: si se disponen de los recursos necesarios, esta acción permite la creación de un goliat. Su sintaxis se muestra en la figura 44.

```
(:action CrearGoliat

:parameters (?b - Base ?j - Juego)

:precondition (and (not (Poblacion_maxima ?j))
                   (Puede_crear_unidades ?b)
                   (>= (minerales) 100)
                   (>= (vespeno) 50)
                   (>= (factorias) 1)
                   (>= (armerias) 1)
                   (<= (+ (poblacion_actual) 4) (poblacion_maxima)))

:effect (and (increase (frames) 300)
             (increase (goliats) 1)
             (decrease (minerales) 100)
             (decrease (vespeno) 50)
             (increase (poblacion_actual) 4)))
```

Figura 44: Acción CrearGoliat

- CrearEspectro: Si se disponen de los recursos necesarios, esta acción permite la creación de un espectro. Su sintaxis se muestra en la figura 45.

```
(:action CrearEspectro

:parameters (?b - Base ?j - Juego)

:precondition (and (not (Poblacion_maxima ?j))
                   (Puede_crear_unidades ?b)
                   (>= (minerales) 150)
                   (>= (vespeno) 100)
                   (>= (puertosespaciales) 1)
                   (<= (+ (poblacion_actual) 4) (poblacion_maxima)))

:effect (and (increase (frames) 300)
             (increase (espectros) 1)
             (decrease (minerales) 150)
             (decrease (vespeno) 100)
             (increase (poblacion_actual) 4)))
```

Figura 45: Acción CrearEspectro

3.4.2.2. Conocimiento de bajo nivel

La representación de bajo nivel basada en objetos presenta la información del entorno del juego en un determinado instante. Esta se construye a través de la información recogida por los sensores y eventos que ofrece BWAPI además de la información que se extraiga de las estructuras que se hayan creado para representar información. Esta información puede ser actualizada total o parcialmente cuando se solicita el estado, cuando un evento ha sido detectado o una acción ha sido ejecutada parcial o totalmente.

Los sensores en el sistema han sido considerados como los elementos que recogen los eventos producidos en el juego, y que BWAPI notifica al sistema, tales como una unidad que ha desaparecido del campo visual o la construcción iniciada de un nuevo edificio. El sistema define un actuador como la representación programada de una acción capaz de interactuar con el objetivo, en este caso StarCraft. En ellos se define el comportamiento a seguir para completar o ejecutar la acción que representan en el entorno además de efectuar los cambios en los predicados y funciones que competen a la acción en el estado de forma total o parcial según se hayan alcanzado las metas de su ejecución.

Representación del estado

El sistema representa el estado en un almacenamiento intermedio que permite almacenar un puntero a cualquier objeto que se desee almacenar mediante un puntero void* enlazado con un tipo correspondiente al tipo de objeto del dominio y un nombre que recibe automáticamente al añadirse al estado si no es especificado otro. Además, se almacenan los predicados y funciones con su formato PDDL y la referencia a los distintos objetos que toman como parámetros o los valores numéricos que posean si son funciones. Sin embargo y pese a esta flexibilidad, sólo se han almacenado punteros a las estructuras que no se encuentran en el juego es decir, no se ha tenido en cuenta para el almacenamiento de las unidades, en las que se ha utilizado el valor ID que BWAPI proporciona y por la cual se pueden recuperar de manera sencilla los punteros a las mismas.

En BWAPI tanto los edificios como las unidades se engloban dentro de la misma clase, unidad. Entre otros datos, esta clase almacena el identificador único de la unidad, la posición en píxeles x e y en la que se encuentra contados a partir de la esquina superior izquierda, sus capacidades y sus valores de puntos de resistencia restante, armadura, ataque y jugador al que pertenece. BWAPI ofrece también la consulta de otros datos relativos al mapa o al juego y que son utilizados por las acciones programadas tal como son los *frames*. Los *frames* son una medida que el funcionamiento del juego provee y que se corresponde con el repintado del juego en pantalla. BWAPI utiliza esta medida para contabilizar el tiempo. Así, por ejemplo, un marine Terran tendría un tiempo de entrenamiento de 300 *frames*. Esta medida es la utilizada también dentro del sistema, ya que cada llamada a las mismas se produce cuando el evento que recoge el paso de un *frame* en el juego es notificado por BWAPI.

Sin embargo, esta representación queda muy limitada para el manejo de objetos con un nivel de abstracción superior que permita, por ejemplo, agrupar distintos tipos de unidades o para permitir un acceso rápido y cómodo a los edificios y unidades cercanos a un punto de interés como puede ser la base inicial. Por ello se creó una estructura que representa una base y que no está contemplada ni en el juego ni en BWAPI. Esta estructura proporciona el conjunto de información necesaria para las acciones implementadas en este proyecto y recibe la denominación de "base". Base representa una agrupación de edificios y unidades alrededor de un edificio principal capaz de crear trabajadores y unos recursos a extraer. La estructura base se ha definido como una estructura general a todas las razas del juego. En ella se han almacenado los identificadores de las unidades y edificios que lo componen. Estos elementos son:

- Una referencia denominada centro del edificio principal, entendido este como el edificio capaz de generar trabajadores y ser almacén en el que los trabajadores depositan los recursos extraídos.

- Una lista de resto de edificios que han sido construidos, siendo el resto de edificios todos aquellos que no son considerados centros de ninguna base.

- Un listado de trabajadores desocupados. En esta lista se encuentran todos aquellos trabajadores a los cuales no se les ha asignado una tarea.

- Un listado de constructores: un listado con los trabajadores que están construyendo algún edificio.

- Un listado de los trabajadores asignados a la recolección de gas.

- Un listado de los trabajadores asignados a la recolección de mineral.

- Un listado de batallón defensivo, entendido como el conjunto de unidades cuya misión es defender la base.

- Un listado de depósitos de minerales: siendo un depósito de mineral perteneciente a una base si la distancia entre éste y el centro de la misma no duplica el valor de la distancia del depósito más cercano al centro tal y como se muestra en la figura 46. Esta distancia es calculada mediante la función de BWAPI para calcular la distancia entre dos unidades y que calcula la distancia euclídea entre las casillas que conformen las unidades.

Figura 46: Distancias a recursos

- Un listado de los géiseres, con las mismas consideraciones que los depósitos de mineral.

- Un listado de unidades que están en proceso de entrenamiento: este listado almacena una referencia a las unidades que actualmente se encuentran en proceso de entrenamiento o se encuentran a la espera de ser entrenadas.

- Un listado del estado de las unidades en construcción: con el fin de permitir la construcción simultánea de unidades inclusive del mismo tipo este listado controla que no existe más de un actuador o acción de entrenamiento de una unidad de ese tipo, vigilando el estado de entrenamiento la unidad.

- Un listado de los edificios en construcción: almacena la referencia a aquellos edificios que actualmente se encuentran en proceso de construcción, es decir que el proceso de construcción por parte de un trabajador no ha finalizado.

- Un listado del estado de los edificios en construcción: similar al listado del estado de las unidades en construcción. Este listado es usado para evitar que varios actuadores de construcción o acciones de construcción operen sobre el mismo edificio y permitir la construcción simultánea de varios edificios tanto del mismo tipo como diferente.

Estos últimos listados son necesarios puesto que los Terran, como cada raza en StarCraft, poseen una manera única de construcción respecto al resto de razas. La forma en que se construye un edificio consiste en:

1. Enviar un trabajador a la posición en la que el edificio se construirá.

2. Iniciar la construcción en ese momento si los requisitos de recursos y de terreno disponible se siguen cumpliendo. En caso de no cumplirse, el trabajador quedará parado en la posición esperando nuevas órdenes.

3. El trabajador estará el tiempo necesario para la construcción operando en las casillas requeridas por el edificio y que se encuentran ya reservadas en la fase anterior.

4. El trabajador queda liberado de la tarea en cuanto el edificio ha sido finalizado.

BWAPI además presenta el inconveniente analizado con anterioridad de tener que determinar cuándo el edificio ha sido completamente construido y que en el caso de no disponer momentáneamente de recursos la orden puede darse como ejecutada, aunque con resultado falso. Finalmente, no todos los edificios de los Terran se construyen de la misma forma. El caso más general es el expuesto en los cuatro pasos señalados anteriormente. Sin embargo, existe un par de excepciones generales al modo de construcción con trabajador:

- Extractor de vespeno: sólo se puede construir en un emplazamiento determinado, los geiseres de vespeno.

- El centro de mando, como en el resto de razas presenta la peculiaridad de no poder ser construido a menos de una distancia mínima de los depósitos de minerales o los geiseres de vespeno.

Además, los complementos de los edificios deben de ser construidos inmediatamente al lado del edificio principal, siendo éste el encargado su construcción. Este edificio no puede construir estos complementos si en ese momento se encuentra entrenando una unidad, realizando una investigación, no se cumplen los requisitos de recursos necesarios o el emplazamiento situado inmediatamente en la esquina inferior derecha del edificio está ocupado por una unidad.

En cuanto a las unidades, su creación como Terran viene determinada por el tipo de unidad que se desea producir, ya que cada unidad es producida únicamente en un edificio y ese edificio generalmente puede producir más de un tipo de unidad distinto. Cada edificio ofrece una cola de construcción de cinco unidades, de forma que si existen recursos disponibles puede solicitarse el entrenamiento de hasta 5 unidades que serán producidas de forma secuencial. En caso de que al terminar el entrenamiento de la unidad no se dispusiera de espacio poblacional suficiente, el entrenamiento de la unidad se quedaría paralizado al final del tiempo requerido para el entrenamiento, retomándose nuevamente en cuanto quede espacio libre para la creación de la unidad. Una vez finalizado el entrenamiento, la unidad aparecerá en una posición contigua más cercana al edificio de creación. BWAPI realiza el posicionamiento de las unidades en tres niveles:

- Píxel: como unidad básica de medida el píxel a máxima resolución.

- Casilla caminable: es una casilla de ocho por ocho píxeles sobre los cuales se dispone de información relativa al movimiento.

- Casilla de construcción: está conformada por cuatro por cuatro casillas caminables, o treinta y dos por treinta y dos píxeles. Reciben esta denominación por que la información de construcción se encuentra a esa resolución.

Representación de las acciones

Las acciones que se han programado dentro del sistema permiten la conversión de una acción solicitada en PDDL en un actuador concreto que opera directamente sobre el juego. Esta conversión es realizada automáticamente por el sistema sobre las acciones que disponga almacenadas y además analizará los nombres de los parámetros indicados en PDDL para servir al actuador los objetos que estos enlazan en la representación intermedia almacenada sobre el juego. El sistema permite además la implementación de dichas acciones de dos formas diferentes, mediante punteros a funciones y mediante una clase que represente la misma.

Acciones por puntero a función

La primera opción para la implementación consiste en el uso de punteros a función con el comportamiento deseado para la acción. Un puntero a una función es un puntero cuyo valor apuntado se corresponde con la dirección en memoria de la función objetivo. Al hacer uso de este puntero se está realizando una llamada a la función enlazada con los objetos que se indiquen como parámetros. La principal ventaja que representa esta forma de ejecución de acciones frente a la segunda forma de implementación de comportamientos es el ahorro de recursos tanto en tiempo al inicializar el actuador que representa la acción cómo al almacenamiento en memoria del mismo al no ser necesaria la instanciación de un objeto para la misma. Estos punteros son almacenados de forma manual dentro del objeto de dominio en el cuál se desean que y los predicados resultantes de la misma pueden ser actualizados de manera automática en función de los objetos recibidos. La sintaxis para añadir una acción de este tipo se compone de las siguientes tareas:

1. Alta de la función y sus efectos en el dominio de aplicación: para la acción cuya declaración es:

   (Recolectar ?t – Trabajador ?dm - Deposito_mineral)

   Y que produce el efecto:

   (Recolectando t dm)

   Es necesario indicar dicha información tal y como se muestra en la figura 47.

```
//Inicialización de la acción Recolectar ¿t - trabajador dm -
//Deposito_mineral
vector<string> tiposObjetos; //Nombres de los tipos de objetos
vector<string> pseudoNombres; // Representación reducida de los
tipos
string accion; //Nombre de la acción
vector<vector<string>> predicados; //Efectos
vector<string> predicado;

//Cabecera de la acción
accion = "Recolectar";
tiposObjetos.push_back("Trabajador");
tiposObjetos.push_back("Deposito_mineral");

pseudoNombres.push_back("t");
pseudoNombres.push_back("dm");

//Efectos de la acción
predicado.push_back("Recolectando");
predicado.push_back("t");
predicado.push_back("dm");
predicado.push_back("true");// Añadir o eliminar el predicado


predicados.push_back(predicado);

dominio->anyadirAccion(accion,   tiposObjetos,   pseudoNombres,
wrapper_botonDerecho, predicados);
```

Figura 47: Declaración de acción por puntero a función

Con esta declaración el sistema es capaz de enlazar los objetos almacenados en el estado para cada llamada recibida en PDDL a la función indicada al almacenarse en el dominio.

2. Implementar la acción objetivo: para la implementación de la acción ha de seguirse una declaración similar a:

```
static std::string wrapper_botonDerecho(void* Sensores, void* objprincipal, void*
params,bool activo);
```

Siendo los parámetros:

- Sensores: El puntero al objeto que representa los sensores y que contiene un acceso al juego.
- Objprincipal: El primer objeto que la acción requiere, en este caso será la unidad ordenada recolectar.
- Params: Un vector que contiene el resto de los objetos que la acción solicitó en la declaración. En este caso, el puntero al objeto unidad del depósito de mineral sobre el cuál comenzar la extracción.
- Activo: Si la acción ya ha sido activada con anterioridad.

Con esta aproximación sólo es necesario realizar las comprobaciones de los prerrequisitos de la acción dentro de la función objetivo y, en caso de cumplirse, ejecutar la acción. Este tipo de acciones sólo se ha comprobado su funcionamiento para la acción anteriormente expuesta y para una acción de test con sintaxis PDDL:

(test ?b - base)

Esta acción simplemente emite un mensaje por la pantalla del juego.

Acciones por instanciación de objetos

En este tipo de comportamientos, las acciones implementadas heredan sus características comunes de una clase definida para tal fin y son instanciadas e inicializadas en el momento en que se recibe la orden de ejecución de la acción. Este enfoque permite el almacenamiento de datos relativos al estado de ejecución propios de la acción de manera independiente. Las acciones que se han programado de esta forma en el sistema permiten la ejecución de las definidas a alto nivel en este proyecto.

Recolección total mineral y vespeno

Este *script* intenta recolectar los depósitos de mineral o vespeno considerados dentro del rango de una base. Para ello se sigue la siguiente fórmula:

Trabajadores_requeridos= (depósitos_mineral_base)*2.

Esta fórmula se debe a que uno o más trabajadores pueden estar extrayendo recursos del mismo depósito u extractor siempre que el resto se encuentre en movimiento. El factor de multiplicación dos es debido a la corta distancia a la que generalmente se encuentran dichos depósitos del centro. En el caso de vespeno este factor es tres dado que los trabajadores pueden esperar su turno de extracción. Si no se dispusiera de trabajadores suficientes, asignaría los trabajadores disponibles catalogados como desocupados de esa base a extraer recursos y notificaría un error para re-planificación al no poderse completar. Si uno de los trabajadores desocupados estuviese siendo entrenado en ese momento, el script se detendría el tiempo suficiente para que dicha unidad terminase su entrenamiento. En el caso del vespeno, ha de comprobar si se dispone de extractor para realizar la tarea, devolver error en caso contrario o esperar si el mismo se encuentra en construcción.

En caso de volver a ser ejecutado, el *script* tendría en cuenta los trabajadores ya asignados anteriormente y continuaría el reparto de trabajadores por el orden en el que se hubiese detenido. Si finalmente se consiguiera la meta de dos trabajadores por depósito de mineral, el *script* incluiría el predicado correspondiente de recolección total de mineral en la base sobre la cual opera y notificaría su correcta ejecución.

En la figura 48 se comprueba un posible reparto inicial que este script puede hacer con los trabajadores iniciales.

Figura 48: Reparto de trabajadores

Construir edificio

Construir edificio es el primero de los dos comportamientos principales programados mediante este enfoque. Este comportamiento tiene como objetivo proporcionar una primera aproximación a la construcción de edificios y unidades para la raza Terran. Por ello, el comportamiento general para ambos se encuentra codificado dentro de una superclase común con el comportamiento de entrenar unidad. Esta superclase basa su comportamiento en una máquina de estados cuyas funciones de transición deberán ser implementadas por cada función que herede de dicha clase y cuyo nombre coincide con el del estado. De esta forma, cada estado implica un pequeño comportamiento en *script*. A continuación en figura 49 se presenta el autómata o máquina de estado que ha sido diseñada para la acción construir edificio.

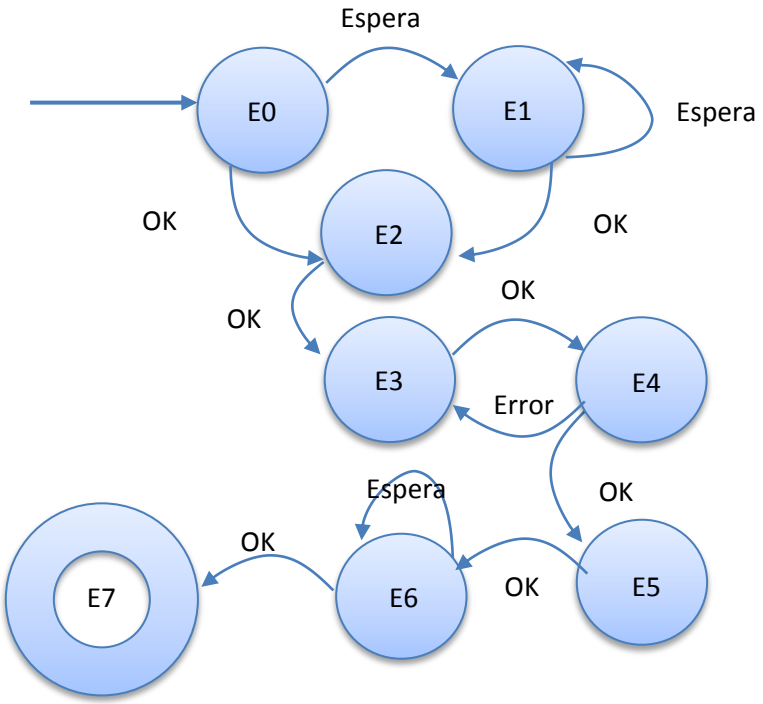

Figura 49: Máquina de estados del script de construcción

Siguiendo este diagrama de estados, la clase que contiene la acción "ConstruirEdificio" sobrescribe las funciones correspondientes a cada uno de ellos de la siguiente forma:

- Estado 0 (E0): es el estado inicial de la máquina de estado e inicia el proceso de construcción, realizando la transición al estado 1.

- Estado 1 (E1): Esperando recursos comprueba que se disponen de los recursos necesarios para la construcción del edificio. En caso contrario, establece una espera de 100 *frames* para esta orden y vuelve a realizar la comprobación.

- Estado 2 (E2): Selección: consiste en seleccionar un trabajador para que realice la construcción del edificio. Este trabajador se obtiene de la lista de trabajadores desocupados de la base destino para trasladarlo a constructores. En caso de no disponer trabajadores en esta categoría, intenta seleccionar el primero de los que se encuentran recolectando mineral.

- Estado 3 (E3): Emplazamiento: en la fase de emplazamiento se intenta establecer qué casilla de construcción se utilizará como referencia para la construcción del edificio. Para realizar la búsqueda del emplazamiento se toma como referencia el centro de la base, es decir el edificio centro de mando de la raza Terran. En BWAPI las casillas de construcción de un edificio comienzan a contarse desde la posición superior izquierda del edificio y se utiliza como casilla de referencia la primera. En el caso del centro de mando, se tiene que sus casillas forman una rejilla de 4x3 casillas de construcción y será la casilla de referencia de este edificio el que se tome como semilla para el algoritmo de emplazamiento. Este algoritmo intenta en cada *frame* encontrar una posición para la construcción del edificio de treinta en treinta para evitar la paralización del juego. Se ha implementado como un algoritmo de búsqueda que realiza sus iteraciones como se muestran en la figura 50.

Figura 50: Búsqueda de emplazamiento para construcción

Dónde:

- Indica la casilla de referencia inicial.
- Indica una posición inválida para un edificio de 3x2 casillas.
- Señala un punto de construcción válido.
- Marca la dirección de comprobación del algoritmo.

- Estado 4 (E4): Ordenar Construcción: una vez que ha sido seleccionado una localización para el edificio comienza el proceso de construcción del edificio indicando al trabajador que comience la construcción en el emplazamiento previamente seleccionado.

- Estado 5 (E5): Identificar Edificio: en cada *frame* recorre la lista de edificios en construcción de la base que previamente hayan sido identificados como tales en los sensores buscando un edificio cuyo tipo coincida con el de la orden de construcción. En caso de encontrar un edificio coincidente, traslada el estado de este edificio en la lista de edificios en construcción observados y establece una espera para el actuador igual al tiempo en *frames* restantes hasta la finalización de la construcción del edificio.

- Estado 6 (E6): Esperando Finalización: este estado comprueba que el edificio ha sido finalizado correctamente y en caso negativo, establece una espera con la siguiente fórmula:

Espera= Tiempo_restante_construcción(Edificio)-5

- Estado 7 (E7): Finalizar: Si el edificio ha sido finalizado, el actuador establece el trabajador constructor como disponible en la base añadiendo el predicado:

(Trabajadores_disponibles Base1)

En caso de que el edificio produjese un aumento en la población disponible, eliminaría el predicado:

(Poblacion_maxima Juego0)

Finalmente, elimina el edificio de la lista de construcción así como su estado en la lista de edificios en construcción observados, lo almacena en la lista de resto de edificios de la base y llama la función para realizar los cambios en los predicados que se hayan indicado al actuador, en caso de que tuviese alguno.

Construir extractor

Como especialización del script comentado anteriormente, construir extractor intenta establecer un extractor de vespeno en uno de los geiseres de vespeno disponibles cerca del centro de mando. Esta clase hereda de la clase ConstruirEdificio comentada anteriormente y que contiene los comportamientos básicos de la máquina de estados para la construcción (figura 51) y sobre-escribe sólo los estados con el *script* comportamiento deseado.

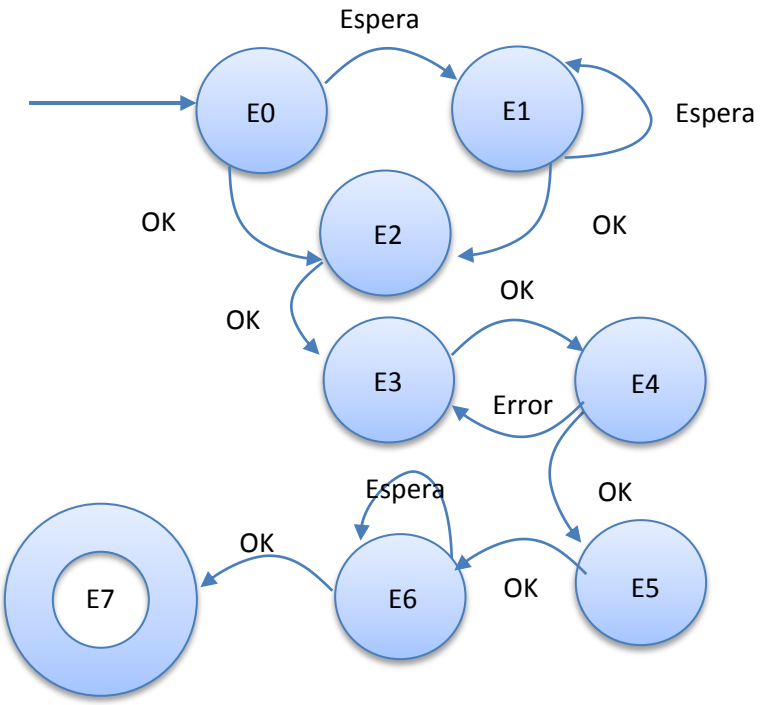

Figura 51: Máquina de estados del script de construcción

Los estados sobre-escritos son los siguientes:

- Estado 3 (E3): Emplazamiento: comprueba si la construcción de un extractor en la posición del geiser de vespeno es posible.

- Estado 5 (E5): Identificar edificio: en caso de ser posible y que su construcción haya sido ordenada, comprueba con otra aproximación si el trabajador constructor esta efectivamente construyendo, introduce el edificio que dicho trabajador está construyendo en la lista de edificios en construcción y establece su estado como observado en la lista de edificios en construcción observados. Esta nueva aproximación es necesaria porque el cambio de tipo de geiser de vespeno no genera un evento de este tipo y, por tanto, no se puede identificar en este el nuevo edificio ni su pertenencia a la base.

- Estado 7 (E7): finalizar: el estado es ampliado para añadir en el estado el siguiente predicado:

(Extractor_vespeno Base1)

Cómo paso previo al estado **E7** del script general.

Entrenar unidad

La acción de entrenamiento de unidades utiliza una máquina de estados similar a la utilizada por el proceso de construcción de edificios. Aunque las acciones que se ejecutan en cada uno de los estados difiere. A continuación se realiza una descripción de cada uno de los estados mostrados en la figura 52:

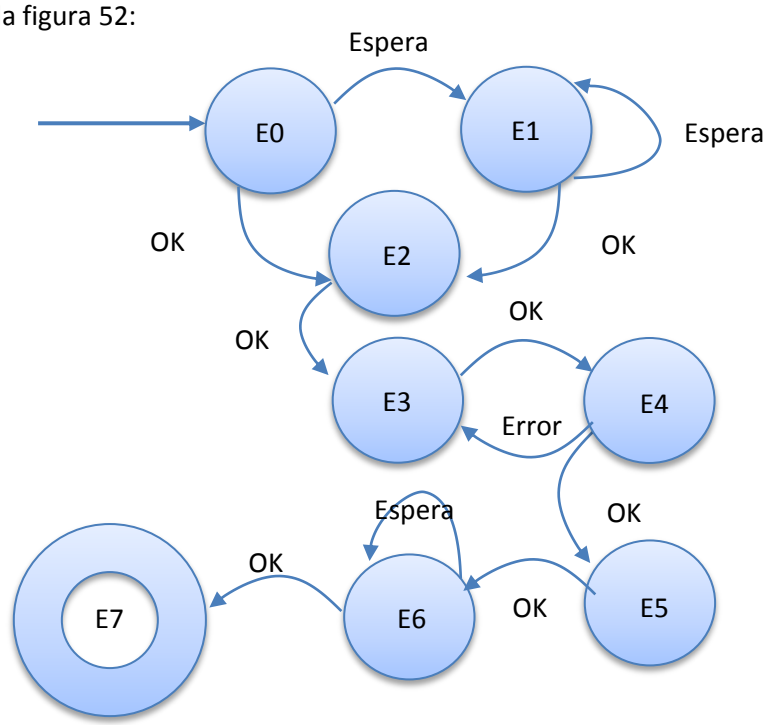

Figura 52: Máquina de estados del script de construcción

El comportamiento que se ha definido para los estados de entrenamiento son los siguientes:

- Estado 0 (E0): Inicial: establece el estado E1.

- Estado 1 (E1): Comprobación de requisitos: al igual que en el script de construcción, en el entrenamiento de una nueva unidad es necesario comprobar que se disponen los recursos básicos para el mismo, mineral y vespeno. A diferencia de la construcción, es necesario comprobar que se dispone del suficiente espacio de población para albergar la nueva unidad. En caso de no disponer de suficiente población, añade en el estado inicial del problema el predicado y devuelve la instrucción para solicitar una re-planificación:

  (Poblacion_maxima Juego0)

  Si no se cumplen los requisitos de recursos básicos, establece una espera de 100 *frames* para el actuador.

- Estado 2 (E2): Selección: en este *script* es necesario determinar qué edificio será el responsable de la construcción de la unidad. Para ello se analiza la lista de edificios de la base comprobando si son capaces de producir el tipo de unidad requerido.

- Estado 3 (E3): Emplazamiento: el emplazamiento viene determinado por el paso anterior; por tanto el comportamiento en este estado es establecer el estado E4.

- Estado 4 (E4): Ejecutar: efectúa la llamada al comando de entrenamiento del edificio para poner en la cola la unidad a crear.

- Estado 5 (E5): Identificar unidad: para realizar la identificación de la nueva unidad, se analiza la lista de unidades en entrenamiento de la base y si el tipo de la unidad, y el edificio en el que se construye coincide con la unidad que ha sido ordenada entrenar y el edificio designado para tal fin siempre que se cumpla que dicha unidad no esté bajo control de otro actuador, se establece como la unidad en producción de este actuador notificándolo en la lista de unidades en entrenamiento observadas. Tras realizar este paso se establece una espera siguiendo;

  Espera=Tiempo_entrenamiento_restante(Unidad)-5

- Estado 7 (E7): Finalizar: en cuanto la unidad ha quedado entrenada, se ejecutan los efectos de los predicados que tenga almacenados el actuador y si la unidad es un soldado se introduce en la lista del batallón defensivo de la base. En caso de que la unidad sea un trabajador, éste se introduce en la lista de trabajadores disponibles de la base y se da de alta en el estado el siguiente predicado:

  (Trabajadores_disponibles Base1)

  Finalmente, se identifica la unidad en la lista de unidades en entrenamiento y la lista de unidades en entrenamiento observadas y se elimina su referencia.

3.5. Flujo general de ejecución

En este apartado se ilustra el comportamiento del sistema. En primer lugar, el comportamiento general del mismo desde el punto de vista de la comunicación y el uso de PELEA se recoge en la figura 53.

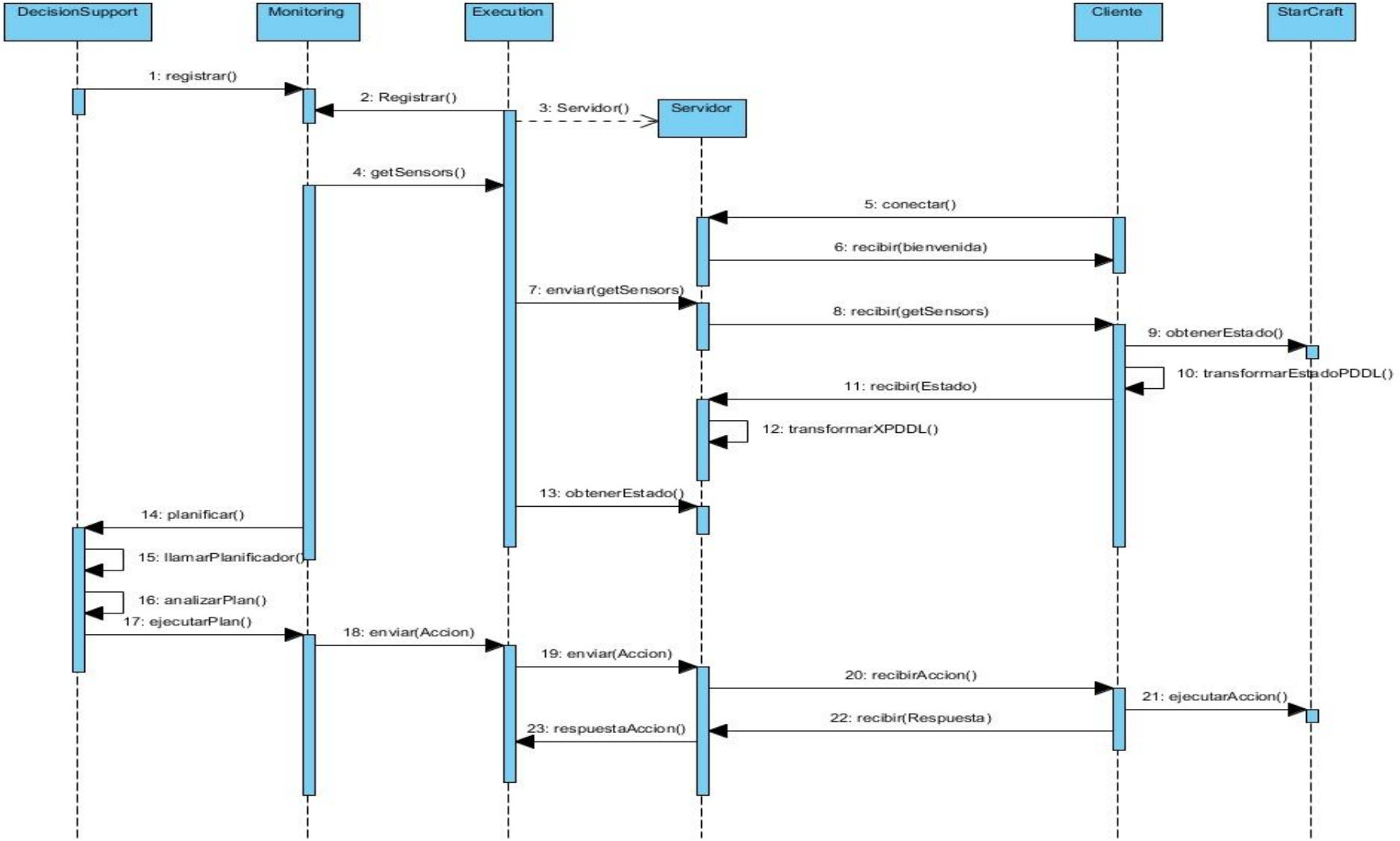

Figura 53: Diagrama de secuencia general

Siendo las llamadas:

1. Registrar(): el módulo *DecissionSupport* se registra dentro del servidor RMI del módulo *Monitoring*.

2. Registrar(): el módulo Execution se registra dentro del servidor RMI del módulo *Monitoring*.

3. Servidor(): *Execution* crea un objeto de la clase Servidor para controlar las comunicaciones.

4. getSensors(): se solicita al módulo *Execution* la información del estado, la información de los sensores.

5. Conectar(): el cliente hace una petición de conexión al servidor.

6. Recibir(Bienvenida): el cliente acepta la conexión y notifica que se encuentra listo para nuevas órdenes.

7. Enviar(getSensors): *Execution* solicita al servidor el envío de un mensaje con la orden para obtener la información del estado.

8. Recibir(getSensors): el cliente recibe la orden para obtener el estado.

9. obtenerEstado(): el cliente analiza la información disponible del juego y actualiza su representación interna del mismo.

10. transformarEstadoPDDL(): con el estado actualizado el cliente genera un fichero en PDDL que contiene el estado.

11. Recibir(Estado): el servidor recibe el envío del mensaje que contiene el estado.

12. trasnformarXPDDL(): con el estado en PDDL se genera un fichero que posteriormente se transforma y lee en el formato de los mensajes que utiliza PELEA, XPDDL.

13. obtenerEstado(): *Execution* que se había mantenido a la espera del mensaje transformado lo solicita al servidor.

14. Planificar(): con la información actualizada del estado el módulo *Monitoring* envía el mensaje para la obtención de un plan.

15. llamarPlanificador(): *DecisionSupport* llama al planificador Metric-FF con el estado y el dominio para generar un plan.

16. analizarPlan(): *DecisionSupport* analiza el resultado del plan generado y lo transforma a XPDDL.

17. ejecutarPlan(): con el nuevo plan generado el módulo *Monitoring* inicia su ejecución.

18. enviarAccion(Accion): el módulo *Monitoring* ordena la ejecución de una de las acciones del plan al módulo *Execution.*

19. Enviar (Accion): la ejecución de una acción comienza cuando el módulo *Execution* de PELEA ordena el envío de la orden con la acción a ejecutar al servidor. Se solicita el envío de la acción con una cadena de texto PDDL extraída del XPDDL que PELEA maneja en su interior.

20. recibir (Acción): el servidor realiza el envío del texto solicitado al cliente.

21. ejecutarAcccion(): el cliente ejecuta la acción ordenada sobre StarCraft y recoge el resultado de su ejecución.

22. Recibir (Respuesta): tras la ejecución de la acción por parte del cliente, éste envía un mensaje con el resultado de la ejecución de la misma. Este mensaje es recibido y decodificado en el hilo de servidor encargado de la recepción.

23. respuestaAccion(): tras comprobar el comando, notifica a *Execution* el resultado de la acción.

En esta versión de PELEA el módulo *Monitoring* no tiene en cuenta el resultado de la acción, por lo que a partir del envío vuelve a realizar una llamada para obtener el estado y tras recibirlo, procede a enviar la siguiente acción a ejecutar. Es el cliente el encargado de forzar la ejecución secuencial de las acciones.

En el caso de que no se use PELEA para la ejecución y control, el diagrama de estados mostrado en la figura 54 muestra su funcionamiento de forma simplificada.

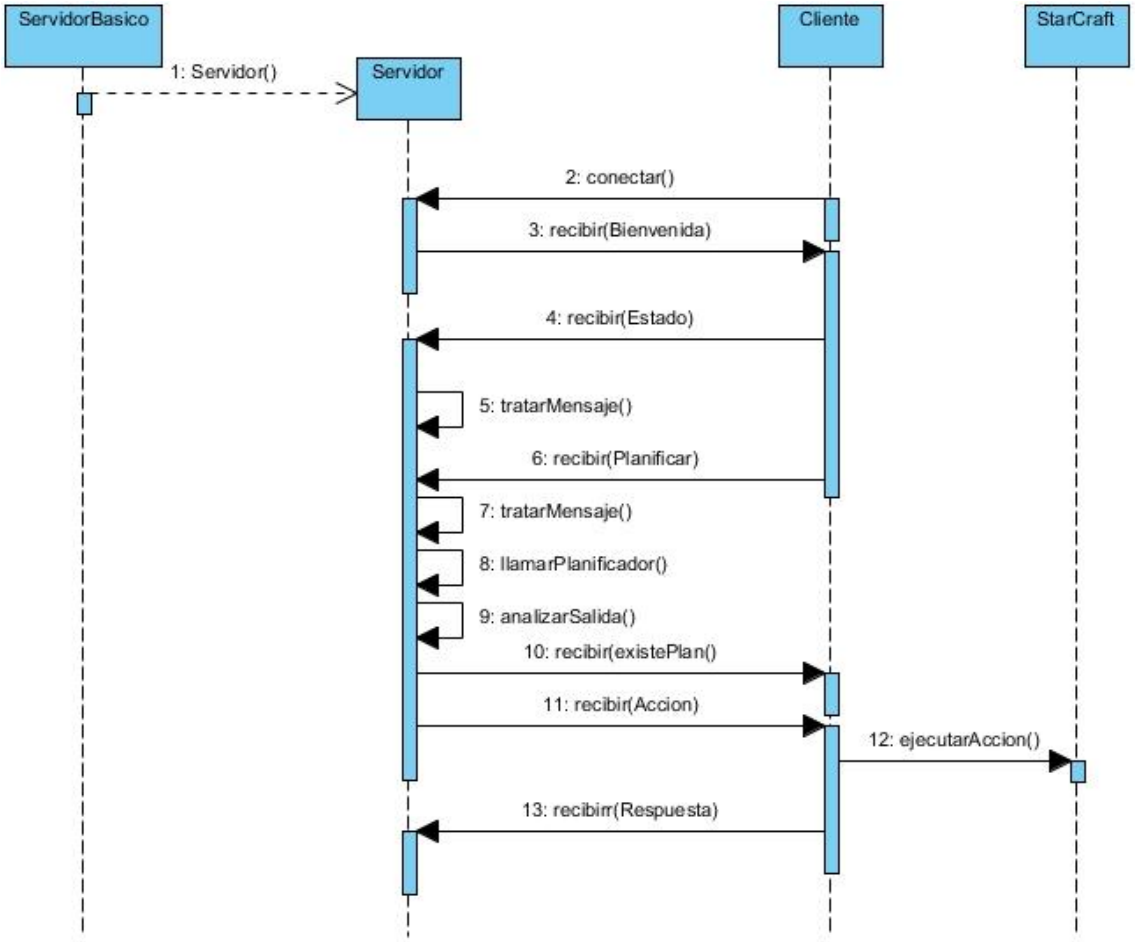

Figura 54: Ejecución sin PELEA

Las llamadas se corresponden con:

1. Servidor: el lanzador de la aplicación crea una instancia de la clase Servidor para controlar las comunicaciones.

2. Conectar: el cliente intenta conectarse al Servidor.

3. Recibir(Bienvenida): el servidor acepta la conexión y envía el mensaje de bienvenida.

4. Recibir(Estado): el cliente envía e indica al servidor que ha recibido un estado.

5. tratarMensaje(): el servidor identifica la orden de Estado como comando público y guarda el estado en un ficero.

6. Recibir(Planificar): el cliente envía al servidor la orden de planificar.

7. TratarMensaje(): el servidor identifica la orden para planificar como un comando público y lo ejecuta.

8. llamarPlanificador(): el comando crea un proceso con la llamada al planificador Metric-FF con el dominio y el estado.

9. analizarSalida(): el servidor realiza un análisis sobre los resultados que el planificador mostraría por pantalla identificando los pasos del plan a ejecutar.

10. Recibir(ExistePlan): si la planificación ha sido satisfactoria, el servidor notifica al cliente que se ha obtenido un plan. Este mensaje se identifica como comando público en el cliente y se marca la señal de plan disponible.

11. recibirAccion(): con el plan recién analizado, el servidor envía la primera acción a ejecutar.

12. ejecutarAccion: el cliente decodifica y ejecuta la acción recibida durante el tiempo que esta requiera.

13. Recibir(Respuesta): con el resultado de la acción se genera un mensaje que contiene su nombre y el resultado de ejecución de la misma y se envía al servidor.

A partir de este momento, si se ha recibido un mensaje “OK” de respuesta a la acción, el servidor envía la siguiente acción a ejecutar. Este comportamiento se repite hasta la finalización del plan.

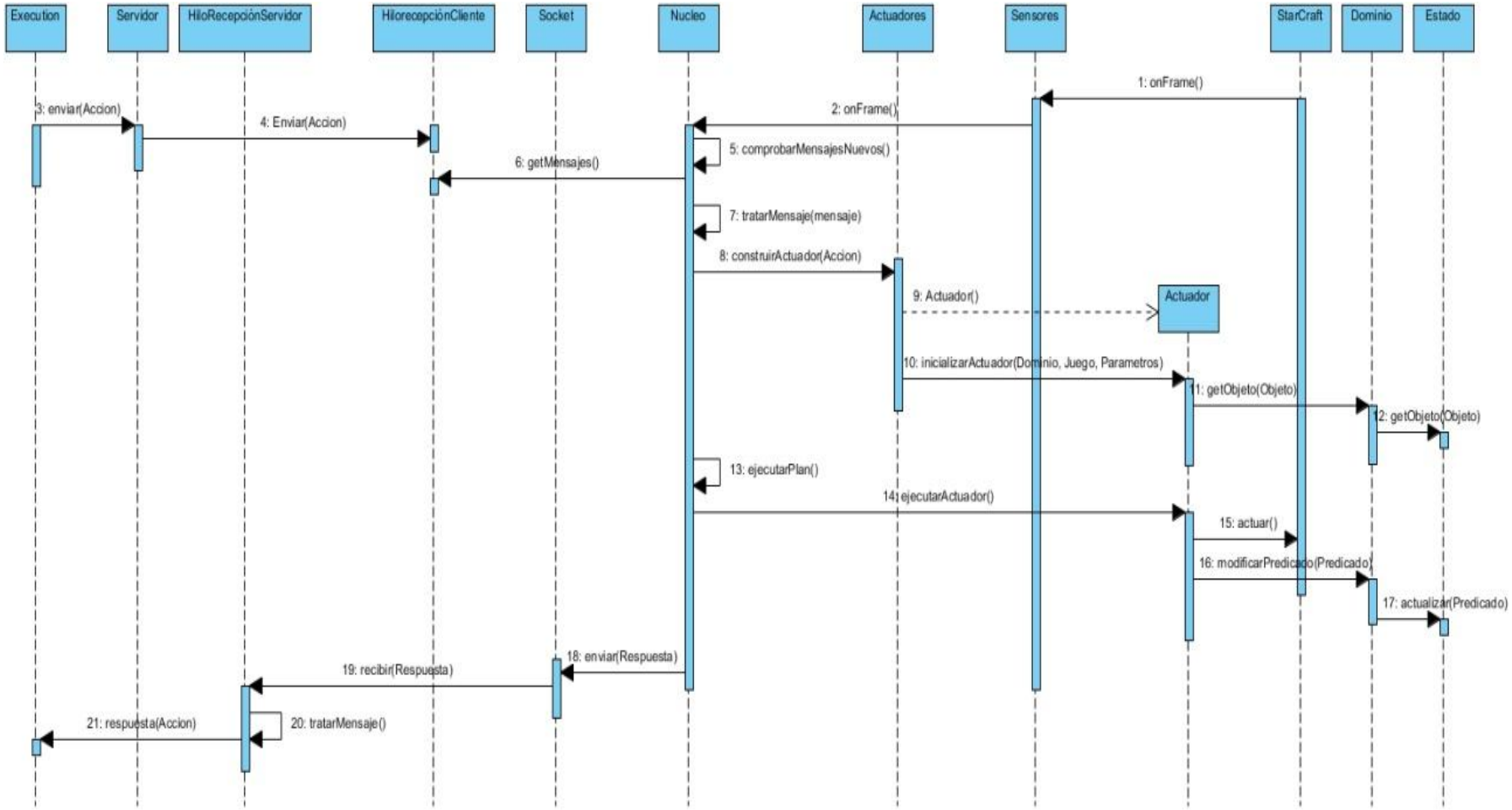

Figura 55: Diagrama de secuencia de ejecución de una acción genérica

Las llamadas mostradas en el diagrama de la figura 55 son:

1. *onFrame():* el juego llama al evento *onFrame*() en cada refresco del juego.

2. *onFrame():* los sensores a su vez notifican el evento al núcleo.

3. enviarAccion()*:* en cualquier momento el servidor envía la orden para la ejecución de una acción en PDDL. Esta acción es decodificada y almacenada en el buzón de mensajes.

4. ComprobarMensajesNuevos: el Núcleo comprueba si hay mensajes nuevos en el buzón.

5. getMensajes(): el Núcleo realiza esta llamada al hilo para obtener los mensajes completos que hayan sido recibidos y se encuentren almacenados en el buzón de recepción.

6. tratarMensaje(Mensaje): con los mensajes recibidos el Núcleo realiza esta llamada para comprobar si se trata de un comando público.

7. construirActuador(Acción): al comprobarse que se trata de una acción, el Núcleo solicita a Actuadores la instanciación de una acción que cumpla con el encabezado recibido si la hubiere. Posteriormente este actuador queda almacenado dentro del plan a ejecutar del Nucleo.

8. Actuador(): Actuadores realiza la instanciación del actuador correspondiente a la acción recibida.

9. inicializarActuador(Dominio, Juego, Parámetros): tras la instanciación, Actuadores notifica al Actuador sobre qué Dominio puede consultar, cuál es su Juego para operar y los parámetros en PDDL de la acción.

10. getObjeto(Objeto): con cada parámetro el Actuador lo solicita al dominio y obtiene el puntero a los mismos.

11. getObjeto(Objeto): Dominio traslada la consulta del puntero al objeto al Estado correspondiente que tenga almacenado.

12. ejecutarPlan(): como paso continuación al tratamiento de los mensajes, Núcleo intenta continuar con el plan que tenga almacenado.

13. ejecutarActuador(): en la ejecución de las acciones el Núcleo llama a esta función cuando es su turno de ejecución.

14. Actuar(): durante uno o más *frames*, el Actuador realiza su trabajo sobre el juego.

15. notificarPredicado(Predicado): si la ejecución ha sido correcta, el actuador notifica al Dominio los cambios que ha de realizar en el Estado.

16. actualizar(Predicado): el Dominio actualiza el Estado con los nuevos cambios en los predicados.

17. enviar(Respuesta): el Actuador devuelve un mensaje con el resultado de la ejecución al Núcleo. Éste realiza el envío del mensaje-respuesta al Servidor.

18. recibir(Respuesta): El servidor recibe y decodifica el mensaje de respuesta.

3.5.1. Control de ejecución del plan

El sistema incorpora dos formas diferentes para el control de la planificación y ejecución de planes: Control por el servidor o por el cliente.

En el control por parte del servidor, el cliente ignora cualquier mensaje de las acciones correspondiente a control de ejecución excepto los *frames* a saltar y es PELEA o el servidor en su defecto quien realiza este control.

En el control por cliente, son las acciones las que determinan los eventos, indicando al núcleo en su respuesta de ejecución si es necesario re-planificar, el método está esperando y no puede ser cancelado o si la ejecución del mismo ha sido correcta y puede ser descartado del plan a ejecutar. Además, se dispone de varias opciones referentes a la ejecución de los planes, la ejecución secuencial o simultánea de opciones y si el sistema acepta acciones duplicadas.

3.5.2. Comunicaciones

Las comunicaciones del sistema se encuentran encapsuladas en dos clases, una para el cliente, y otra para el servidor. En el caso del cliente, la clase que contiene el comportamiento posee los elementos necesarios para la gestión de la comunicación básica:

- Socket de envío de datos.
- Socket de recepción de datos con escucha continúa.
- Socket para recepción de conexiones entrantes.
- Buzón de mensajes identificados.

En ella se desensamblan los mensajes de los marcadores que use el servidor para la comunicación, y se almacenan los mismos hasta que el núcleo los requiera.

En el servidor, este comportamiento se encuentra dentro de la clase encargada también del tratamiento de los mensajes y ejecución de los comandos públicos.

Capítulo 4: Evaluación

En este apartado se presentan los resultados de las pruebas realizadas al sistema para comprobar su correcto funcionamiento. Para ello se detallará el entorno en el que se van a realizar las pruebas, se describirá el formato de las mismas, y se presentará un resumen con las conclusiones que se han extraído al realizar las pruebas.

4.1. Descripción del entorno de pruebas

Las pruebas se han realizado en el equipo de sobremesa en el cual se ha instalado una máquina virtual con el sistema operativo Windows XP instalado en ella. La máquina virtual contiene el cliente de StarCraft y el juego. El juego se encuentra instalado en su ruta por defecto y ChaosLauncher se ha ubicado en el escritorio. Como configuración especial se ha establecido el adaptador puente como opción de la máquina virtual para permitir las comunicaciones.

El servidor de PELEA se ha integrado con los módulos del mismo y se ha desplegado en el escritorio del ordenador de sobremesa en el sistema operativo KUbuntu (16) (17). El planificador Metric-FF ha sido compilado para dicho sistema operativo y se ha integrado dentro de las carpetas que contiene PELEA. Las rutas que se han configurado para los ficheros de dominio y estado se corresponden con la localización del ejecutable del planificador Metric-FF.

4.2. Descripción de las pruebas

A continuación se realiza una descripción de los atributos que serán utilizados para la descripción de las pruebas que han sido realizadas para probar el correcto funcionamiento del sistema que ha sido desarrollado para este proyecto.

- Objetivo: establece cuales son los puntos que tiene la prueba como meta.
- Explicación: contiene una descripción de la prueba.
- Tareas: se define la lista de pasos seguidos para realizar la prueba, organizados en servidor y cliente.
- Resultados: se muestran los mensajes o efectos producidos tras la ejecución de la prueba. Se incluirá una referencia al vídeo demostrativo de su funcionamiento.
- Análisis de los resultados: se analizan los efectos producidos durante la ejecución incluyendo aquellos que hacen referencia al rendimiento.
- Problemas encontrados: se describen los problemas encontrados tras la repetición de la prueba.
- Conclusiones: se explica el éxito o fracaso de la prueba.

4.3. Pruebas realizadas

A continuación se van a presentar las pruebas generales que se han realizado al sistema y que comprobarán exclusivamente el correcto funcionamiento del mismo.

1. Registro de sucesos (log)

Objetivo	Esta prueba tiene como objetivo principal comprobar que el sistema almacena correctamente un registro con los sucesos iniciales del sistema. Como objetivos secundarios se comprobará que el sistema reconoce las unidades iniciales propias del jugador, es capaz de generar una base adicional a la base vacía con los datos iniciales así como de aceptar funciones definidas y mostrar si ha ocurrido un error en la conexión.
Explicación	Se ejecutará el juego a través de ChaosLauncher sin estar el servidor de PELEA operativo y se comprobará en el fichero generado que los objetivos se cumplen.
Tareas	• Tareas en el cliente 1. Desplegar el fichero DLL en la carpeta del juego. 2. Ejecutar el juego a través de ChaosLauncher. 3. Iniciar una partida.
Resultados	El sistema registra la lectura de las opciones de configuración del mismo, crea las estructuras necesarias para la comunicación y funcionamiento y muestra los eventos relativos a los objetivos de la prueba (18).
Análisis de los resultados	El sistema cumple con los requisitos de la prueba sin suponer un coste notable añadido en el rendimiento del juego.
Problemas encontrados	No se han encontrado problemas en la ejecución de esta prueba.
Conclusiones	El cliente ofrece una información correcta de inicialización del sistema. Esta información adquiere una mayor relevancia para la depuración y comprobación de las pruebas realizadas a continuación.

Tabla 72: Prueba de registro de sucesos (log)

2. Configuración y conexión

Objetivo	En esta prueba se evalúa una configuración diferente a la que por defecto incluye el cliente. Se comprueba que el cliente es capaz de identificarla, procesarla y conectarse correctamente al servidor.
Explicación	Se configurará el cliente para que sea capaz de conectarse a la IP del área local en la cual se esté ejecutando el servidor y se establecerá un puerto diferente. Posteriormente se comprobará en el fichero de registro que se realiza la conexión y que se recibe el mensaje de bienvenida del servidor.
Tareas	• Tareas en el servidor 1. Lanzar el script para la ejecución del servidor. 2. Obtener la dirección en la cual se ejecuta el servidor. • Tareas en el cliente 1. Establecer el puerto de comunicaciones común a 4004 en el fichero de configuración. 2. Establecer la dirección IP del cliente a la obtenida en el servidor.
Resultados	El fichero de registro muestra una conexión correcta con el servidor y que recibe de este el mensaje de bienvenida (19) .
Análisis de los resultados	Tras establecerse las estructuras iniciales y detectarse las unidades como en la primera prueba se comprueba cómo la conexión se efectúa correctamente y el sistema muestra el mensaje de bienvenida sin bloqueos en el juego.
Problemas encontrados	Tras repetir la prueba, se ha podido encontrar en raras ocasiones un posible error de conexión por parte del cliente.
Conclusiones	• El sistema permite una conexión correcta con el servidor salvo raras excepciones que no afectan al comportamiento del servidor. Estos problemas de conexión derivan de la forma de conexión del propio sistema operativo. • El sistema permite conectarse a una IP asignada sin necesidad de recompilar código. Esto es de gran utilidad para ejecuciones en diferentes equipo o cuando el mismo equipo cambia su dirección de red.

Tabla 73: Prueba de configuración y conexión

3. Envío de mensajes servidor/ cliente

Objetivo	Esta prueba tiene como objetivo comprobar la correcta comunicación entre el ejecutable del servidor básico y el cliente de StarCraft.
Explicación	Se hará uso del comando 'enviar' en la consola del servidor seguido del mensaje "Hola mundo". Este mensaje deberá ser correctamente mostrado por la consola de mensajes del juego StarCraft.
Tareas	• Tareas en el servidor 1. En la consola del servidor se escribirá el comando "enviar". 2. En la consola se escribirá "Hola mundo". • Tareas en el cliente 1. Comprobar que el mensaje "Hola mundo" aparece en la consola del juego.
Resultados	El sistema muestra correctamente el mensaje en la pantalla de juego (20).
Análisis de los resultados	El envío de mensajes se realiza de forma fluida y sin complicaciones.
Problemas encontrados	No se han encontrado problemas para esta prueba.
Conclusiones	• El envío de mensajes del servidor al cliente se efectúa de forma correcta. Esta prueba es de vital importancia ya que se comprueba que el enlace de comunicaciones de envío por parte del servidor y recepción por parte del cliente se realiza de forma correcta y sin producir bloqueos en el cliente. • Además se comprueba que el encapsulamiento de mensajes establecido para las comunicaciones es decodificado correctamente en el cliente y codificado de igual forma en el servidor.

Tabla 74: Prueba de envío de mensajes servidor/ cliente

4. Ejecución de comando público de cliente desde el servidor

Objetivo	En esta prueba mostrará que el sistema es capaz de reconocer comandos enviados por el servidor y actuar conforme al comportamiento definido por el mismo. Además se comprueba que las acciones definidas por el usuario mediante retro-llamadas quedan correctamente registradas dentro del dominio.
Explicación	Se hará uso del comando 'enviar' en la consola del servidor seguido del mensaje "/mostrar'dominio". Este mensaje mostrará el listado de acciones que se encuentren registradas en el dominio.
Tareas	• Tareas en el servidor 1. Escribir "enviar" en la consola del servidor. 2. Escribir "/mostrar'dominio" en la consola. • Tareas en el cliente 1. Comprobar que las acciones son mostradas en el juego.
Resultados	El sistema muestra correctamente la orden recibida y la ejecución del comando (21).
Análisis de los resultados	El sistema permite la ejecución correcta y sin retardos de los comandos solicitados por el servidor.
Problemas encontrados	No se han encontrado problemas para esta prueba.
Conclusiones	El servidor es capaz de ejecutar las órdenes establecidas como públicas dentro del cliente sin complicaciones. Esta prueba es muy importante porque indica que el servidor es capaz de ejecutar tareas adicionales a la planificación en el cliente.

Tabla 75: Prueba de ejecución de comando público de cliente desde el servidor

5. Envío de mensajes cliente/servidor

Objetivo	En esta prueba se demuestra que el cliente es capaz de enviar mensajes al servidor y que este a su vez es capaz de decodificarlos correctamente.
Explicación	Se hará uso del comando "/planificar" que envía un mensaje al servidor con la cadena "/m:planificar". Se podrá comprobar que el servidor recibe correctamente la cadena de texto.
Tareas	• Tareas en el servidor 1. Comprobar en la consola que se recibe correctamente el mensaje de planificar. • Tareas en el cliente 1. Escribir como mensaje del jugador "/planificar" dentro del propio juego.
Resultados	El servidor recibe y decodifica correctamente el mensaje (22).
Análisis de los resultados	El envío del mensaje se produce de forma inmediata.
Problemas encontrados	No se han encontrado problemas para esta prueba.
Conclusiones	Al igual que su homóloga servidor/cliente, esta prueba es de vital importancia para constatar que la escucha y decodificación por parte del servidor se realiza correctamente así como el envío y codificación de mensajes por parte del cliente.

Tabla 76: Prueba de envío de mensajes cliente/servidor

6. Ejecución de comando público de servidor

Objetivo	El objetivo de esta prueba es comprobar que el servidor es capaz de reconocer y ejecutar comandos públicos enviados por el cliente.
Explicación	Se hará uso del comando "/planificar" que envía un mensaje al servidor con la cadena "/m:planificar". Se podrá comprobar que el servidor recibe correctamente la cadena de texto y ejecuta una planificación con el estado que se encuentre almacenado.
Tareas	• Tareas en el servidor 1. Comprobar en la consola que el comando ha sido ejecutado. • Tareas en el cliente 1. Escribir como mensaje del jugador "/planificar" dentro del propio juego.
Resultados	El servidor ejecuta correctamente la acción y realiza la planificación utilizando el planificador externo (23).
Análisis de los resultados	El juego continúa en funcionamiento mientras se realiza la planificación, por lo que no existe retardo en el mismo.
Problemas encontrados	Si el estado almacenado no es compatible con el dominio que usa el planificador, es decir, se usan hechos, funciones u objetos no definidos en el dominio este da error al planificar.
Conclusiones	El servidor es capaz de reconocer y ejecutar comandos marcados como públicos. Esta prueba es muy importante ya que con ella se puede indicar al servidor que órdenes ejecutar si es el cliente el que controla la ejecución de la planificación.

Tabla 77: Prueba de ejecución de comando público de servidor

7. Ejecución local servidor

Objetivo	Esta prueba tiene como objetivo comprobar que el servidor puede ejecutar órdenes programadas como privadas.
Explicación	Se hará uso del comando 'contar en la consola del servidor seguido del mensaje "Hola mundo". Este comando devolverá la cantidad de caracteres de la cadena por pantalla del servidor.
Tareas	• Tareas en el servidor 1. Escribir "contar" en la pantalla del servidor. 2. Escribir "Hola mundo" en la pantalla del servidor.
Resultados	El sistema muestra la longitud correcta de la cadena: 10 (24).
Análisis de los resultados	El sistema responde a las peticiones privadas sin interferir en otras funciones como la espera para la conexión.
Problemas encontrados	No se han encontrado problemas para esta prueba.
Conclusiones	El servidor es capaz de programar comportamientos no accesibles desde clientes públicos. Esto permite una mayor flexibilidad en su comportamiento ya que de esta forma se puede configurar manualmente.

Tabla 78: Prueba de ejecución local servidor

8. Ejecución local cliente

Objetivo	Se comprobará en esta prueba que el cliente es capaz de ejecutar comandos establecidos como privados.
Explicación	Se hará uso del comando "/show players" en la consola del juego. Este mensaje ordenará que se muestren los jugadores pertenecientes a cada fuerza aliada en el juego.
Tareas	• Tareas en el cliente 1. Escribir "/show players" en la consola del juego.
Resultados	El juego muestra correctamente los jugadores de cada fuerza de la partida (25).
Análisis de los resultados	Los comandos ejecutados de esta forma son ejecutados dentro del propio hilo del juego, por lo que si se prueba otro comportamiento complejo es posible que el rendimiento del mismo se vea afectado.
Problemas encontrados	Si se prueban comandos que requieran mucho cómputo, puede influir en el rendimiento del juego.
Conclusiones	El sistema permite la ejecución de comandos privados con posibles mermas en el rendimiento del juego si el comando requiere de cómputo elevado. Sin embargo, al igual que los comandos privados del servidor, resulta útil para cambiar datos de configuración o de comportamiento manualmente durante el juego.

Tabla 79: Prueba de ejecución local cliente

9. *Prueba de pérdida de conexión servidor*

Objetivo	En esta prueba se comprobará que el servidor es capaz de aceptar nuevas conexiones tras la pérdida de una anterior.
Explicación	Se desconectará manualmente el cliente para forzar la pérdida de conexión y posteriormente se relanzará, comprobando que la nueva conexión con el servidor se ha establecido sin problemas.
Tareas	• Tareas en el cliente 1. Tras ejecutarlo normalmente, cerrar el juego. 2. Relanzar el juego y comprobar que las comunicaciones se realizan con normalidad.
Resultados	El sistema admite sin limitación las nuevas conexiones tras la pérdida de las mismas sin que el servidor se vea afectado (26).
Análisis de los resultados	El servidor no ve afectado su comportamiento, pero su estado interno, si se estuviera ejecutando algún comando continuará como si el juego no hubiese sufrido pérdida.
Problemas encontrados	No se ha encontrado problemas en esta prueba.
Conclusiones	El servidor admite un cierto grado de tolerancia a fallos por parte del cliente o de la red no siendo necesario cerrar y volver a ejecutar el mismo y sin verse afectado su rendimiento.

Tabla 80: Prueba de pérdida de conexión servidor

10. *Prueba de acciones individuales: recolección total mineral*

Objetivo	En esta prueba se comprueba que el sistema es capaz de ejecutar acciones programadas correctamente.
Explicación	Se hará uso del comando 'enviar' en la consola del servidor seguido del mensaje "(RecoleccionTotalMineral Base1)". Este mensaje deberá ser correctamente tratado y la acción ejecutada hasta los límites que se han establecido en la misma.
Tareas	• Tareas en el servidor 1. Escribir 'enviar' en la consola del servidor. 2. Escribir el mensaje "(RecoleccionTotalMineral Base1)" en la consola. • Tareas en el cliente 1. Comprobar que la acción es ejecutada en el juego.
Resultados	A los trabajadores iniciales se les ordena recolectar los depósitos de mineral cercanos a la base, uno por cada depósito y la acción concluye indicando que ha sido inconclusa, solicitando re-planificación ya que no se dispone de los trabajadores suficientes para recolectar completamente los depósitos (27).
Análisis de los resultados	El cliente ejecuta la tarea sin mostrar retardo. El comportamiento es el esperado dados los requisitos iniciales.
Problemas encontrados	No se han encontrado problemas para esta prueba.
Conclusiones	El sistema es capaz de ejecutar acciones individuales correctamente así como notificar los resultados de su ejecución. Esta prueba es de vital importancia ya que permite constatar que el cliente es capaz de operar con el juego de manera directa dada una acción programada y un estado definido.

Tabla 81: Prueba de acciones individuales: recolección total mineral

11. Prueba de obtención de estado

Objetivo	En esta prueba se confirma que el sistema es capaz de devolver la información de un estado en formato PDDL correcto.
Explicación	Se hará uso del comando 'enviar' en la consola del servidor seguido del mensaje "/getSensors". Este mensaje deberá ser correctamente tratado y ejecutado de tal forma que el cliente realice una transformación de la representación interna del estado a PDDL correcta y que esta transformación es enviada al servidor.
Tareas	• Tareas en el servidor 1. Escribir 'enviar' en la consola. 2. Escribir "/getSensors" en la consola. 3. Comprobar que el estado llega correctamente. • Tareas en el cliente 1. Comprobar que el mensaje ha sido tratado, el estado transformado y enviado.
Resultados	El sistema envía el estado en formato PDDL de forma correcta. Este es identificado como tal dentro del servidor (28).
Análisis de los resultados	El envío del estado se realiza de forma correcta y no se observa una merma en el rendimiento del cliente. Además, incluye datos de funciones de forma correcta.
Problemas encontrados	No se han encontrado problemas para esta prueba.
Conclusiones	El cliente es capaz de mantener una representación interna del estado del juego correcta así como transformarla a formato PDDL. Esta prueba es de vital importancia ya que serán los cambios en el estado los que indicarán a PELEA la fase de ejecución de las acciones o, si el control lo lleva el cliente, disponer de datos de estado sobre los cuales el planificador pueda trabajar.

Tabla 82: Prueba de obtención de estado

12. Prueba de cambio de estado

Objetivo	Se comprobará en esta prueba que el sistema es capaz de recoger cambios producidos en el estado del juego.
Explicación	Se hará uso del comando 'enviar' en la consola del servidor seguido del mensaje "/getSensors" y se tomará nota de los valores plasmados en el estado. Transcurridos unos segundos y tras haber solicitado entrenar un trabajador, se repetirá el proceso comprobando los cambios en el estado. Se comprobará que el juego es capaz de realizar cambios en los valores de las funciones que recogen la cantidad de minerales, *frames* o vespeno.
Tareas	• Tareas en el servidor 1. Escribir 'enviar' en la consola del servidor. 2. Escribir el mensaje "/getSensors" . 3. Repetir los pasos 1 y 2 transcurridos unos segundos. 4. Comprobar que el estado ha cambiado. • Tareas en el cliente 1. Enviar a recolectar minerales a los trabajadores iniciales. 2. Crear un trabajador tras el primer mensaje "getSensors".
Resultados	El cliente actualiza correctamente los valores de las funciones cuyos valores se actualizan de forma continua. Estos cambios quedan correctamente reflejados en el contenido PDDL generado tras transformar el estado interno del juego (28).
Análisis de los resultados	El sistema es capaz de cambiar los datos del estado sin afectar al rendimiento general del juego. Los nuevos valores son correctamente expresados en el instante en que el estado es solicitado.
Problemas encontrados	No se han encontrado problemas para esta prueba.
Conclusiones	El cliente es capaz de actualizar correctamente los valores de las funciones cuyo contenido se actualiza cuando el estado es pedido al cliente. Con esta prueba se demuestra que el sistema es capaz de reflejar cambios producidos o bien por acciones o por el transcurso del tiempo en el propio juego.

Tabla 83: Prueba de cambio de estado

13. Recolección total de vespeno

Objetivo	En esta prueba se verifica que la acción programada para la extracción de vespeno funciona correctamente.
Explicación	Se hará uso del comando 'enviar' en la consola del servidor seguido del mensaje "(RecoleccionTotalVespeno Base1)". Este mensaje deberá ser correctamente tratado y ejecutado, tanto si es posible ejecutarlo como si no es posible por que alguno de los requisitos no se cumpla.
Tareas	• Tareas en el servidor 1. Escribir en la consola del servidor el comando 'enviar'. 2. Escribir en la consola "(RecoleccionTotalVespeno Base1)". • Tareas en el cliente 1. Verificar que los trabajadores son ordenados recolectar vespeno.
Resultados	La acción se ejecuta correctamente, solicitando re-planificación si no se encuentra extractor de vespeno disponible o trabajadores para realizar la tarea (29).
Análisis de los resultados	Los efectos en el juego se producen de manera inmediata y sin retardo.
Problemas encontrados	No se han encontrado problemas para esta acción en el escenario de prueba por defecto.
Conclusiones	El comportamiento del script es adecuado a lo que se requería. Esta acción es importante puesto que provee de uno de los recursos extraíbles necesarios para la construcción y entrenamiento de muchos edificios y unidades además de las investigaciones.

Tabla 84: Prueba de recolección total de vespeno

14. Construcción de un edificio

Objetivo	Esta prueba verifica que el script de construcción de edificios funciona correctamente y que éste es capaz de emplazar y construir edificios independientemente de sus costes.
Explicación	Se hará uso del comando 'enviar' en la consola del servidor seguido del mensaje "(ConstruirDeposito Base1)". Este mensaje deberá ser correctamente mostrado por la consola de mensajes del juego StarCraft y deberá poder ser comprobado en el fichero de log.
Tareas	• Tareas en el servidor 1. Escribir en la pantalla del servidor el comando 'enviar'. 2. Escribir el mensaje "(ConstruirDeposito Base1)". • Tareas en el cliente 1. Verificar la construcción en el juego del edificio.
Resultados	El cliente emplaza el edificio para su construcción y controla su construcción correctamente además de dar de alta o baja los predicados de forma correcta en el estado (30).
Análisis de los resultados	El script presenta cierto retardo en *frames* cuando existen problemas para el emplazamiento del edificio, por ejemplo, cuando varios trabajadores o edificios se encuentran situados en las posiciones colindantes al centro de mando.
Problemas encontrados	Si el trabajador queda rodeado por otras unidades o edificios, el *script* puede fallar.
Conclusiones	La funcionalidad básica de la construcción de edificio se encuentra cubierta si bien esta cobertura no es óptima y bajo determinadas circunstancias consideradas excepcionales, puede fallar. El funcionamiento de esta acción es muy importante ya que con ella se puede construir gran cantidad de edificios y, a través de ellos, entrenar unidades.

Tabla 85: Prueba de construcción de un edificio

15. Construcción de extractor

Objetivo	Esta prueba tiene como objetivo demostrar que es posible emplazar un extractor de vespeno en un géiser.
Explicación	Se hará uso del comando 'enviar' en la consola del servidor seguido del mensaje "(ConstruirExtractorVespeno Base1)". Este mensaje deberá ser correctamente mostrado por la consola de mensajes del juego StarCraft y deberá poder ser comprobado en el fichero de *log*.
Tareas	• Tareas en el servidor 1. Escribir en la pantalla del servidor el comando 'enviar'. 2. Escribir el mensaje "(ConstruirExtractorVespeno Base1)". • Tareas en el cliente 1. Verificar la construcción en el juego del extractor.
Resultados	El cliente construye correctamente el extractor en el géiser de vespeno (29).
Análisis de los resultados	Esta acción se ejecuta sin problemas de rendimiento y de forma satisfactoria.
Problemas encontrados	No se han encontrado problemas para esta prueba.
Conclusiones	El *script* permite la construcción correcta de un extractor de vespeno necesario para la acción recolección total de vespeno.

Tabla 86: Prueba de construcción de extractor

16. Entrenamiento de trabajador

Objetivo	En esta prueba se demuestra que el sistema es capaz de crear y gestionar nuevos trabajadores.
Explicación	Se hará uso del comando 'enviar' en la consola del servidor seguido del mensaje "(CrearTrabajadores Base1 Juego0)". Este mensaje deberá ser correctamente mostrado por la consola de mensajes del juego StarCraft y deberá poder ser comprobado en el fichero de *log*.
Tareas	• Tareas en el servidor 1. Escribir en la pantalla del servidor el comando 'enviar'. 2. Escribir el mensaje "(CrearTrabajadores Base1 Juego0)". • Tareas en el cliente 1. Verificar la ejecución de la acción.
Resultados	Se entrena una unidad de tipo trabajador correctamente en el centro de mando, posteriormente esta es transferida a trabajadores desocupados en la base (31).
Análisis de los resultados	La acción se ejecuta sin retardo en el juego.
Problemas encontrados	No se han encontrado problemas para el escenario básico, si bien puede producirse un error si la cola de construcción del centro está completamente ocupada.
Conclusiones	El sistema es capaz de crear nuevos trabajadores de forma satisfactoria con ciertas restricciones en cuanto a la cantidad simultánea de los mismos. Esta acción es muy importante ya que son los trabajadores los encargados de la recolección y la construcción de edificios.

Tabla 87: Prueba de entrenamiento de trabajador

17. Entrenamiento de otra unidad

Objetivo	Esta prueba tiene como objetivo determinar que el sistema es capaz de crear unidades de distinto tipo en otros edificios.
Explicación	Se hará uso del comando 'enviar' en la consola del servidor seguido del mensaje "(CrearMarine Base1 Juego0)" cuando se disponga de cuarteles para poder llevarse a cabo. Este mensaje deberá ser correctamente mostrado por la consola de mensajes del juego StarCraft y deberá poder ser comprobado en el fichero de log.
Tareas	• Tareas en el servidor 1. Escribir en la pantalla del servidor el comando 'enviar'. 2. Escribir el mensaje "(ConstruirCuarteles Base1)". 3. Escribir en la pantalla del servidor el comando 'enviar'. 4. Escribir el mensaje "(CrearMarine Base1 Juego0)" tras la finalización de la construcción de cuarteles. • Tareas en el cliente 1. Verificar el entrenamiento de la unidad.
Resultados	La unidad es entrenada correctamente y los valores de la función que contabiliza su número actualizados. Además, traslada la nueva unidad al pelotón defensivo de la base de forma correcta (32).
Análisis de los resultados	El funcionamiento de esta acción es correcto con las mismas restricciones que crear trabajador.
Problemas encontrados	No se han encontrado problemas para esta prueba.
Conclusiones	El sistema es capaz de producir diferentes unidades en los edificios que permiten su entrenamiento de forma correcta. Esta acción es muy importante puesto que es la que permite la creación de todas las unidades si su entrenamiento es posible.

Tabla 88: Prueba de entrenamiento de otra unidad

18. Prueba de cambio de funciones

Objetivo	En esta prueba se verifica que los valores de las funciones son actualizados tanto por el transcurso del tiempo como por cambios producidos al entrenar una unidad o construir un edificio.
Explicación	Se hará uso del comando 'enviar' en la consola del servidor seguido del mensaje "/getSensors". Posteriormente se esperará unos segundos, se entrenará un trabajador y se repetirá el proceso para comprobar cambios en las funciones "frames" y "minerales". Este mensaje deberá ser correctamente mostrado por la consola de mensajes del juego StarCraft y deberá poder ser comprobado en el fichero de *log*.
Tareas	• Tareas en el servidor 1. Escribir en la pantalla del servidor el comando 'enviar'. 2. Escribir el mensaje "/getSensors". 3. Repetir los pasos 1 y 2 tras ordenar la creación de un trabajador. 4. Comprobar los cambios. • Tareas en el cliente: 1. Crear un trabajador tras la obtención primera del estado.
Resultados	Los cambios en las funciones son reflejados correctamente.
Análisis de los resultados	Las funciones cuya actualización no requiere de continuas consultas al juego se realiza de forma inmediata y aquellas que requieren acceso instantáneo al juego son obtenidas en el momento de generar el estado sin producir retardos en el resto del juego (28).
Problemas encontrados	No se han encontrado problemas para esta prueba.
Conclusiones	El sistema permite la actualización correcta de valores de funciones. Esta prueba es importante puesto que permite comprobar que dominios que contengan funciones son aceptados y operativos para el sistema.

Tabla 89: Prueba de prueba de cambio de funciones

19. Prueba de acciones por puntero de funciones

Objetivo	En esta prueba se verifica que los comportamientos programados mediante punteros a función son ejecutados correctamente.
Explicación	Se hará uso del comando 'enviar' en la consola del servidor seguido del mensaje "(test Base1)". Este mensaje deberá ser correctamente mostrado por la consola de mensajes del juego StarCraft y deberá poder ser comprobado en el fichero de *log*.
Tareas	• Tareas en el servidor 1. Escribir en la pantalla del servidor el comando 'enviar'. 2. Escribir el mensaje "(test Base1)". • Tareas en el cliente 1. Verificar que el sistema escribe por pantalla un saludo indicando que se ha reconocido y ejecutado el actuador por puntero a función.
Resultados	El sistema muestra correctamente el mensaje (33).
Análisis de los resultados	No se aprecia retardo por la ejecución de la acción.
Problemas encontrados	Los datos introducidos para las llamadas por puntero a función han de ser muy precisos debido a que la obtención de otros datos adicionales del juego o del estado requiere de operaciones adicionales. No se almacenan datos adicionales del estado del actuador para próximos frames.
Conclusiones	El formato de acciones por puntero a función funciona correctamente, sin embargo, es menos práctico e intuitivo que las acciones programadas con instanciación de objetos.

Tabla 90: Prueba de

20. Prueba de planificación con PELEA

Objetivo	El objetivo de esta prueba es comprobar que el sistema es capaz de hacer uso de la herramienta PELEA para la planificación y ejecución de los planes complejos.
Explicación	Se hará uso del *script* programado para el lanzamiento de PELEA y se iniciará una partida indicando al cliente que el control es externo. Se verificará la correcta ejecución del plan.
Tareas	• Tareas en el servidor 1. Inicializar el script con los módulos de PELEA. • Tareas en el cliente 1. Verificar que la configuración del control de ejecución lo tiene PELEA. 2. Comprobar la ejecución del plan.
Resultados	El sistema es capaz de proporcionar los estados cuando PELEA lo requiere, enviar al cliente las acciones individuales a ejecutar, ejecutar dichas acciones y notificar a PELEA el resultado de las mismas (34).
Análisis de los resultados	El rendimiento de esta forma de operación es ligeramente inferior a la planificación sin PELEA tal y como puede comprobarse en los instantes iniciales hasta que el sistema empieza a ejecutar el plan generado. Sin embargo, no se produce ningún tipo de colisión al ejecutarse las acciones individualmente y permite futuros trabajos con el planificador PELEA que incluyan acciones ejecutadas en paralelo.
Problemas encontrados	Si la opción de ejecución en paralelo está activada para el cliente, esta versión reducida de PELEA envía acciones de forma continua, por lo que el cliente trata de ejecutar en paralelo las mismas pudiendo producirse una falta de precisión en el resultado esperado. Además, el lector de estados de PELEA no está completo, por lo que no es capaz de aceptar ciertas características del lenguaje tales como el uso comparativo de variables o la asignación de valores entre éstas.
Conclusiones	El sistema permite el uso de PELEA de forma correcta incluyendo la comunicación con el mismo. Esta prueba es de vital importancia puesto que permite ceder el control de la ejecución de las acciones a una herramienta ajena al propio sistema, permitiendo desarrollar acciones más simples en el cliente y dominios más complejos en el servidor.

Tabla 91: Prueba de planificación con PELEA

21. Prueba de planificación sin PELEA

Objetivo	En esta prueba se verifica que el sistema es capaz de realizar una planificación y ejecución de un plan limitado sin PELEA.
Explicación	Se verificará que el control de la ejecución se encuentra en el cliente. Durante la prueba se verificará el envío de estado, el tratamiento del mismo por el servidor, la ejecución del planificador y el análisis de sus resultados, el envío del plan resultante y la ejecución del mismo. En este caso al no disponerse de PELEA para la planificación, será la clase ServidorBásico la encargada de realizar la llamada al planificador y de extraer el plan resultante cuando el cliente se lo indique.
Tareas	• Tareas en el servidor 1. Comprobar el plan obtenido y la ejecución del mismo. • Tareas en el cliente 1. Verificar el correcto funcionamiento de las acciones. 2. En caso de encontrarse algún constructor atrapado por otras unidades u edificios, retirar las mismas.
Resultados	El sistema permite un control limitado sobre la ejecución de planes de forma autónoma a PELEA al quedar ese comportamiento programado de forma explícita en las acciones del cliente (35).
Análisis de los resultados	El rendimiento obtenido en esta prueba es ligeramente superior al obtenido durante las pruebas con PELEA, tal y como puede comprobarse desde el inicio hasta la primera acción ejecutada. Sin embargo, si las acciones son ejecutadas en paralelo puede aparecer colisión de las mismas. El consumo de recursos en memoria RAM es reducido al no tener que usar la máquina virtual de Java. El juego no presenta retardos en su comportamiento.
Problemas encontrados	No se han encontrado problemas particulares para esta prueba en el funcionamiento del sistema. En cuanto a los *scripts*, sin embargo, se ha observado que la construcción puede dejar bloqueada una unidad entre distintos edificios.
Conclusiones	El sistema permite una planificación y ejecución de planes limitada pero funcional. Este comportamiento es de utilidad si se desea usar el sistema al margen de PELEA para dominios no extensos.

Tabla 92: Prueba de planificación sin PELEA

22. Prueba de ejecución de planes de gran longitud sin PELEA.

Objetivo	En esta prueba se verifica que es posible la ejecución de planes de longitud elevada.
Explicación	Se generarán unos objetivos que requieran un plan de longitud elevada, de más de ciento cincuenta acciones y se verificará que el sistema es capaz de ejecutarlo correctamente. Al igual que en la prueba anterior, ServidorBasico se encargará de planificar cuando el cliente se lo ordene.
Tareas	• Tareas en el servidor 1. Verificar la longitud del plan generado y el correcto envío de las acciones. • Tareas en el cliente 1. Verificar la correcta ejecución de las acciones.
Resultados	El sistema ejecuta de forma secuencial de forma correcta las acciones indicadas (35).
Análisis de los resultados	Las acciones son ejecutadas sin producirse retardo en el juego.
Problemas encontrados	Si las acciones tienen consideración secuencial y el sistema tiene activado el formato en paralelo, esta secuencialidad puede verse alterada al poderse ejecutar otra acción con menor coste de forma adelantada. Además, es posible que una acción solicite una re-planificación aun estando en ejecución otras. En este escenario, las acciones en ejecución continúan su ejecución pudiendo producirse acciones repetidas que no fueron planeadas si al sistema se le indica que puede tener acciones repetidas. En caso de no aceptar acciones repetidas y recibir dos veces la misma acción, por ejemplo entrenar un trabajador, la segunda y siguientes quedan descartadas y no son ejecutadas.
Conclusiones	El sistema permite la ejecución de acciones en paralelo con limitaciones.

Tabla 93: Prueba de ejecución de planes de gran longitud sin PELEA.

4.4. Resultado de las pruebas

En este apartado se van a exponer los resultados obtenidos tras la ejecución de las pruebas tanto con el uso de PELEA como sin él comparando ambos.

4.4.1. Resultados generales

En lo referente al establecimiento de conexión, el sistema es capaz de configurar la dirección y el puerto al que el cliente se ha de conectar correctamente en una situación normal. Además, las comunicaciones posteriores entre ambos sistemas se efectúan de forma correcta sin provocar ningún retardo en las partes. (19)

En cuanto al control de ejecución, la aplicación es capaz de controlar el flujo de ejecución siempre y cuando se incluyan estos comportamientos de forma programada dentro de los *script* de las acciones. Sin embargo, este control es muy básico y obliga a ciertas restricciones tales como que no se hallen repetidas las acciones o que los planes generados no tengan una longitud muy elevada, ya que las acciones se ejecutarán de forma paralela y, en caso de necesitar re-planificación en alguna, las otras acciones pueden continuar en ejecución.

Tal y como se muestran en los resultados de las pruebas de planificación con y sin PELEA (34) (35), el sistema es capaz de ejecutar las acciones que se le demandan de forma estable durante un periodo de tiempo considerable, 39 minutos, sin identificarse problemas que impidieran su funcionamiento por un periodo más extenso.

4.4.2. Generación de estado

Se ha podido comprobar en la prueba 12 que el sistema es capaz de generar un estado, que este es correctamente transmitido al servidor, que, en caso de disponer de PELEA, lo transforma para que esta herramienta pueda hacer uso de este y en caso de no disponer de un sistema basado en Inteligencia Artificial, es capaz de identificar los comandos programados para el control y actuar en consecuencia.

Se ha comprobado también que los parámetros que reciben las acciones son los especificados por la acción recibida desde el servidor, por lo que el sistema identifica y almacena correctamente los objetos que el estado requiera utilizar independientemente del tipo de objeto que este sea. Esto implica que el sistema permite usar constructos abstractos no disponibles directamente de la información de los sensores otorgando de esta forma una mayor semántica al estado almacenado dejando al desarrollador la decisión de que datos sobre los mismos quedan plasmados en el momento de la generación del estado.

También ha quedo comprobado que el estado es posible modificarlo a través de cambios en el juego producidos por eventos en el mismo durante el transcurso de la partida o bien por una actualización de los valores de funciones del estado cuando se invoca la función que genera el estado actualizado del juego en PDDL, es decir, cuando una acción ha sido ejecutada correctamente y sus post-condiciones han de actualizarse en el estado. Estas actualizaciones también se pueden realizar en cualquier parte de la ejecución de los comportamientos por lo que una ejecución parcial de los mismos puede programarse.

4.4.3. Comportamientos

En cuanto a la ejecución de comandos y órdenes introducidas por un usuario del sistema, se ha comprobado que existen los cuatro apartados definidos para la inclusión de nuevos comportamientos y que estos funcionan correctamente. Además, si estos comandos conllevan actualizaciones en el estado interno del cliente, estas actualizaciones se llevan a cabo correctamente. Estos comandos son devueltos satisfactoriamente por las acciones programadas para el control del juego, tanto por retro-llamada como por orientación a objetos. (21) (23) (24) (25)

En cuanto a las acciones programadas, se ha podido comprobar el correcto funcionamiento de los mismos dentro de las limitaciones incluidas en los mismos tanto cuando las acciones se ejecutan en paralelo como secuencialmente (27). Sin embargo, en paralelo presentan unas limitaciones que pueden hacer que unas acciones interfieran en otras, por lo que es tarea del planificador determinar qué acciones serán ejecutadas en paralelo.

Finalmente, el uso de la planificación automática posee una ventaja añadida sobre el desarrollo de la lógica del comportamiento en el cliente, ya que si se desarrollan comportamientos muy simples dentro del cliente, es PELEA en todo momento el encargado del control del estado del juego, liberando a los comportamientos programados en el cliente de la necesidad de lógica adicional para comprobar la correcta ejecución de los mismos.

Esto posee una contrapartida puesto que es necesaria la generación de un estado más detallado y una mayor cantidad de envíos del estado actualizado que, considerando que el sistema envía cada vez el estado al completo, supone un mayor gasto en comunicaciones. No obstante, este problema puede ser mitigado si se mejora la transmisión del estado al envío de los cambios producidos con respecto al anterior envío.

4.4.4. Variaciones en el entorno y errores

Se ha comprobado que el sistema es capaz de observar las variaciones en el entorno y almacenarlas en el estado. Además, las acciones en ejecución son capaces de percibir esos cambios y de actuar en consecuencia pidiendo una re-planificación o enviando un mensaje de error. (28)

Estos errores no tienen en cuenta los producidos por la ejecución en paralelo de las mismas, por lo que es necesario introducir ese comportamiento en los *scripts* para que sean capaces de actuar en consecuencia. Este comportamiento introduce un grado superior de complejidad en los *scripts* con respecto al uso de PELEA ya que se deben tener en cuenta las limitaciones de la ejecución en paralelo mencionadas además de tener en cuenta el coste superior que conlleva una necesidad mayor de re-planificaciones completas al requerirse una re-planificación por cada excepción de las acciones que se encuentren ejecutadas en paralelo. Es decir, si se dispone de diez acciones ejecutándose en paralelo y tres de ellas encuentran un error por tener carácter secuencial, el cliente notifica tres veces la necesidad de una re-planificación. Esta complejidad puede ser reducida haciendo uso de la versión reducida de PELEA ya que es PELEA quien se encarga de notificar que acciones se ejecutan y cuando hacerlo, liberando al cliente de esa tarea.

El uso de PELEA sin embargo, tiene una contrapartida en consumo de recursos, pero asegura el buen funcionamiento sin incluir control innecesario en el sistema. A pesar de ello, y como puede observarse en el vídeo enlazado en los resultados de la prueba de planificación con PELEA y sin PELEA, esta versión reducida de PELEA es capaz de ofrecer un comportamiento similar al proporcionado por el sistema de forma básica. Con las ventajas expuestas anteriormente que pueden aplicarse en un futuro.

Capítulo 5: Gestión del proyecto

En este apartado se va a detallar el ciclo de vida seguido para la ejecución del proyecto, las distintas fases de las cuáles se ha compuesto y se describirán los medios empleados para su realización y el coste del mismo en el presupuesto.

5.1. Ciclo de vida

Para la realización de este proyecto se ha optado por un ciclo de vida de desarrollo en espiral considerado adecuado para el mismo dada la experiencia en el lenguaje de desarrollo y la complejidad del mismo. Este modelo es mostrado en la figura 56 y se caracteriza por la realización de iteraciones sobre las distintas fases del proyecto facilitando la inclusión de nuevas funcionalidades en cualquiera de las iteraciones.

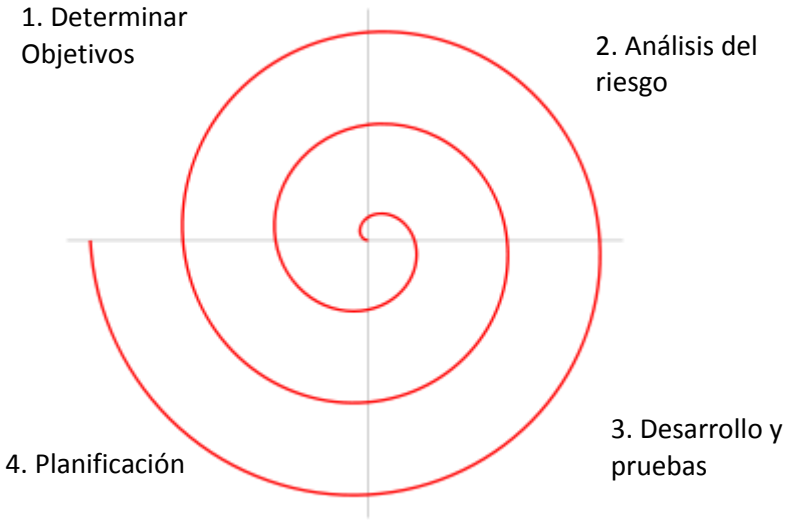

Figura 56: Ciclo de vida en espiral

Este modelo es utilizado precisamente por su capacidad para incluir nuevos cambios definidos permitiendo la generación de nuevas versiones consistentes y estables en cada iteración. Cada una de estas iteraciones produce un sistema con mayor funcionalidad al concretarse más los prototipos generados en la versión anterior y la inclusión de las nuevas características pedidas por el cliente. Las fases que se han seguido son:

1. Determinar objetivos: durante esta etapa se definen los objetivos y requisitos de la iteración además de la identificación de los riesgos y las alternativas de solución. Durante esta fase se realiza la búsqueda de información relacionada tanto como el dominio de aplicación del proyecto como de las tecnologías utilizadas para su desarrollo

2. Análisis del riesgo: en esta etapa se analizan las posibles causas de los posibles inconvenientes y eventos que surjan durante la iteración y las consecuencias que pueden producir.

3. Desarrollo y pruebas: durante esta fase se define las arquitecturas utilizadas describiendo el diseño arquitectónico y el detallado. Tras esta definición, se traslada la información generada a código tangible capaz de cumplir con los requisitos identificados y las limitaciones analizadas con las arquitecturas diseñadas para tal fin. Tras la implementación, se comprueba el correcto funcionamiento del sistema.

4. Planificación: se evalúa el sistema creado y que este cumple con lo especificado para la iteración y se planifica la siguiente actividad.

   El diagrama de Gantt que recoge las iteraciones planificadas se muestra en la figura 57.

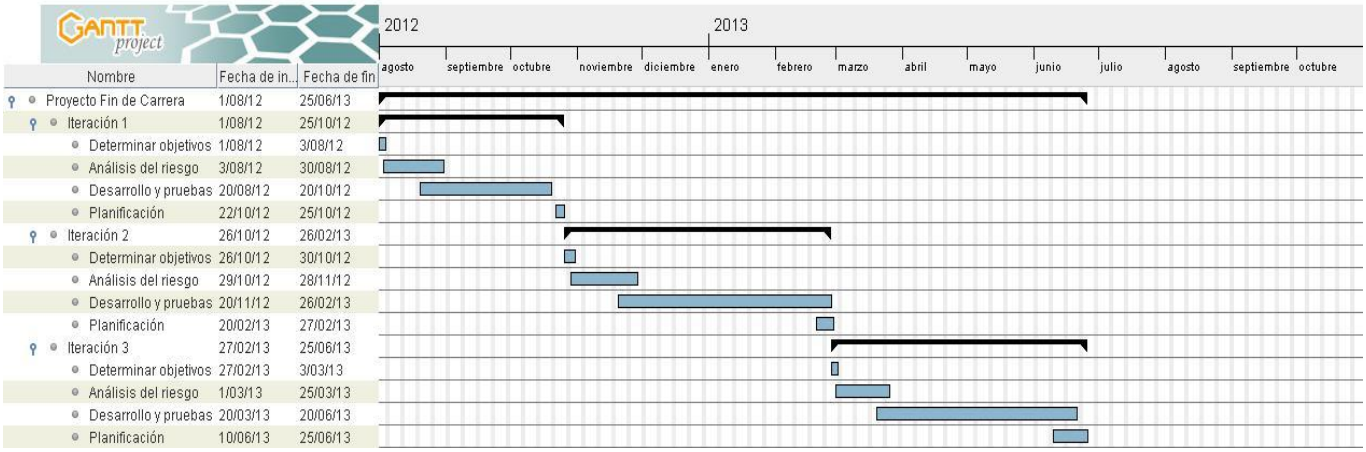

Figura 57: Diagrama de Gantt

5.2. Medios empleados

En este apartado se va a proceder a enumerar los recursos utilizados para el desarrollo del proyecto, tanto a nivel de software como de hardware.

5.2.1. Hardware

Se va a proceder a enumerar los equipos y periféricos utilizados para el desarrollo.

- Portátil Toshiba Tecra:
  - Objetivo: proporcionar un punto para acceder remotamente al ordenador de sobremesa y permitir una navegación fluida. Requerido por el uso inicial de PELEA sobre Apache™®.
  - Procesador: Intel® Core"2 Duo CPU P9300 @ 2.26GHz x 2.
  - Memoria: 1,8 Gb DDR 2 800 Mhz.
  - Tarjeta gráfica: Mobile 4 Series Chipset Integrated Graphics 256Mb.

- Ordenador de Sobremesa:
  - Objetivo: permitir el desarrollo central del proyecto, siendo a la vez servidor y cliente para PELEA y StarCraft.
  - Procesador: Intel® Pentium(R) Dual CPU E2180 @ 2.00GHz × 2.
  - Memoria: 3,9 Gb DDR 667 Mhz.
  - Tarjeta gráfica: Intel® G33 256Mb.
- Disco externo: 250,1 GB.
- Otros: Router inalámbrico.

5.2.2. Software

- KUbuntu 12 04: Ubuntu es un sistema operativo gratuito y libre basado en la distribución Linux, Debian. Proporciona actualizaciones cada 6 meses. KUbuntu es Ubuntu que usa como entorno de escritorio la versión KDE.
- Windows XP Professional SP3: Windows es un sistema operativo ampliamente utilizado con diferentes versiones que se adaptan a lo que el usuario necesite.
- VirtualBox (36): es una solución de virtualización que emula el funcionamiento de una CPU y permite la instalación en la simulación de otros sistemas operativos. El objetivo de este programa es permitir ejecutar el juego que requiere del sistema operativo Windows además de permitir su desarrollo en diferentes equipos.
- NetBeans (37): es un IDE, o entorno de desarrollo integrado, que permite el desarrollo de código en diferentes lenguajes de programación. Se ha utilizado para los componentes Java que interactúan directamente con PELEA.
- Microsoft Visual C++ 2008 Express Edition: es un IDE de Microsoft que permite el desarrollo y compilación de código C++ para Windows. C++ es un lenguaje de programación con soporte para la programación por objetos que está basado en C. Es un lenguaje dependiente de la plataforma. Se ha utilizado para crear el sistema.
- BWAPI: se trata de un API, o interfaz de programación de aplicaciones, permite la interacción con StarCraft. Es un proyecto abierto que requiere de un inyector para realizar su función. Es el punto de partida del proyecto y una parte muy importante del mismo, ya que será el encargado de recoger los eventos del juego, proporcionarlos al código del proyecto y ejecutar las ordenes que este le despache.

-  PELEA: es un sistema de planificación y ejecución. En su versión pública funciona sobre Apache Tomcat®™ y bajo un sistema Unix. Está desarrollado por:
  - *GRPS-AI. Group of Reasoning on Planning and Scheduling* - Artificial Intelligence / GTI-IA Grupo de Tecnología Informática-Inteligencia Artificial. Universitat Politècnica de València.
  - *Planning and Learning research Group* Universidad Carlos III de Madrid
  - Grupo de Sistemas Inteligentes Universidad de Granada.

- OpenVNC (38): se trata de una aplicación que permite el control remoto del escritorio. Se ha utilizado para el control del ordenador de sobremesa desde el portátil.

-  MicrosoftWord y Microsoft PowerPoint: ambas aplicaciones pertenecen a la suite de Microsoft Office y su finalidad es el desarrollo de la documentación y la presentación de la misma.

-  Notepad++ (39): es un programa de edición de textos con una amplia gama de lenguajes de programación reconocidos. Se ha utilizado para visualizar el resultado XML que el programa realizó en una de sus fases además de para la visualización rápida del código existente.

-  StarCraft: es un juego de estrategia en tiempo real o RTS por sus siglas en inglés de reconocimiento mundial en el que tres razas diferentes luchan por sobrevivir.

-  ChaosLauncher: es una aplicación que permite el inyectado de DLL en el juego StarCraft.

-  Java: Java es un lenguaje de programación interpretado que permite su ejecución en diferentes plataformas. Esto es debido a que se ejecuta en una máquina virtual que interpreta el código. PELEA está escrito en Java, por ello se optó por usar este lenguaje y su máquina virtual en el desarrollo del código que interactúa con PELEA.

- Metric-FF: es un sistema de planificación independiente de dominio que opera con PDDL. Está escrito en C y es una extensión del planificador FF capaz de almacenar un determinado conjunto de variables numéricas en los estados. FF es un sistema de planificación heurística que estima las distancias al objetivo y considera que las acciones no son independientes entre sí. FF combina la búsqueda sistemática con el método de búsqueda local de escalada consistente en la búsqueda de una acción que reduzca el coste con respecto al padre (12).

5.3. Presupuesto

En este apartado se detalla el coste presupuestado para el desarrollo del presente proyecto de fin de carrera desglosando del presupuesto en personal, material, y resumen final de todos los gastos.

5.3.1. Personal

A continuación se muestra una tabla con el coste total de personal en función de las horas dedicadas en un rol específico según el BOE Núm. 174 del sábado 21 de julio de 2012 Sec. III. Pág. 52649-52652

Categoría	Dedicación (horas/persona)	Coste (horas/persona)	Coste Total (€)
Jefe de proyecto	70	24,67	1726,9
Investigador	25	25,27	1263,5
Analista	100	24,19	4838,0
Programador	400	22,09	8836,0
		Total:	**16664,4**

Tabla 94: costes de personal

El coste total bruto por personal asciende a 16664,4 €, dieciséis mil seiscientos sesenta y cuatro euros con cuatro céntimos.

5.3.2. Material

A continuación se muestran los costes derivados en software y hardware:

Nombre	Coste (€)	Periodo de amortización (meses)	Duración (meses)	Tiempo desde la compra (meses)	Coste
KUbuntu 12 04	0,00	6	10	12	0,00
Windows XP Professional SP3	159,59	60	10	40	26,60
VirtualBox	0,00	60	10	36	0,00
NetBeans	0,00	60	10	36	0,00
Microsoft Visual C++ 2008 Express Edition	0,00	60	10	10	0,00
BWAPI	0,00	60	10	10	0,00
PELEA	0,00	60	10	10	0,00
OpenVNC	0,00	60	10	10	0,00
Microsoft Office	139,00	60	10	10	23,17
Notepad++	0,00	60	10	10	0.00
StarCraft Broodwar	14,99	60	10	10	2,50
ChaosLauncher	0,00	60	10	10	0,00
Metric-FF	0,00	60	10	10	0,00
				Total:	52,27

Tabla 95: costes de software

Nombre	Coste (€)	Periodo de amortización (meses)	Duración (meses)	Tiempo desde la compra (meses)	Coste
Portátil Toshiba Tecra	990,00	60	10	46	165,00
Ordenador de Sobremesa	450,00	60	10	55	37,50
Disco externo	50,00	60	10	36	8,33
Router inalámbrico	0,00	60	10	12	0,00
				Total:	210,83

Tabla 96: costes de hardware

El coste total asociado al software es de 52,27€, cincuenta y dos euros con veintisiete céntimos. El coste del hardware es de 210,83€, doscientos diez euros con ochenta y tres céntimos. El coste total por materiales es de 271,43€, doscientos setenta y un euros con cuarenta y tres céntimos.

5.3.3. Resumen

A continuación se muestra el resumen del proyecto, de los costes totales brutos del mismo, el coste del IVA y el coste total.

- Nombre del proyecto: Implementación de una arquitectura para el desarrollo de comportamientos para StarCraft apoyado en PELEA.
- Autor: Javier Márquez Colás.
- Duración: Diez meses a tiempo parcial.

Concepto	Coste
Personal	16664,4
Material	271,43
Coste indirecto (10%)	1693,59
Total bruto	18629,42
TOTAL (IVA al 21%)	**22541,60**

Tabla 97: resumen de costes

EL coste final del proyecto es de 22541,60€, veintidós mil quinientos cuarenta y un euros con sesenta céntimos.

Capítulo 6: Conclusiones y trabajos futuros

6.1. Conclusiones

En este apartado se detallan las conclusiones obtenidas durante el desarrollo del proyecto, tanto generales como las referentes a los objetivos planteados para el mismo en el capítulo 1 de este proyecto.

6.1.1. Conclusiones generales

El objetivo del proyecto tal y como se detalla en el apartado objetivos, consiste en la implementación de una arquitectura para el desarrollo de *bots* para StarCraft apoyado en PELEA, siguiendo además otros requisitos impuestos por la competición de AIIDE tales como que el juego no se vea ralentizado por el *bot* desarrollado.

Tal y como se demuestra en los vídeos enlazados en las pruebas realizadas, este objetivo se ha cumplido ya que es posible generar un *bot* de forma sencilla cuyo comportamiento se base en planificación. Este *bot* es generado mediante el uso de pequeños *script5* que pueden ser programados mediante dos técnicas diferentes y que permiten la reutilización del código.

Además, se ha utilizado una versión reducida de PELEA que reduce los costes operativos ligados al uso de Apache-Tomcat®™, resultando en una drástica mejora en la cantidad de memoria RAM utilizada al no ser necesario el despliegue del servidor para su funcionamiento.

Tras cumplir con el objetivo principal, se ha podido comprobar los resultados aplicados en el juego de la planificación automática como aproximación a la toma de decisiones. En este aspecto, se ha concluido que es una alternativa viable incluso para dominios complejos como es StarCraft. Esta conclusión se ha llegado tras comprobar los siguientes puntos:

- Velocidad: si bien en un principio este punto se tomó como problemático debido a la inoperancia de las unidades del jugador durante la planificación en realidad no ha resultado tal debido a la velocidad del planificador utilizado, capaz de generar planes de longitud elevada en tiempos de un segundo. Esto demuestra que la planificación pese a suponer una pérdida inicial en el tiempo de juego redunda en un beneficio final al disponerse de un plan completo a ejecutar.

  Este punto si puede ser una pérdida en batallas, si no se dispone de información previa, puesto que en menos de un segundo se decide el destino de la misma. En este aspecto, el sistema ofrece la posibilidad de incluir comportamientos reactivos, por lo que mientras que el planificador realiza su tarea, se pueden programar comportamientos que tomen decisiones limitadas en función de la información disponible.

- Adaptabilidad: la planificación ofrece una versatilidad que los comportamientos basados en reglas o *scripts* no pueden ofrecer sin una gran complejidad en el código de los mismos. Esta complejidad ha quedado muy reducida en los *scripts* programados dentro de la aplicación, que han quedado simplificados a un comportamiento particular para un problema pequeño, dejando al planificador la tarea de decisión sobre qué paso ha de ejecutarse a continuación.

Estas ventajas no están completas, ya que la planificación a través de PELEA ofrecerá comportamientos más complejos capaces de controlar la correcta ejecución de los *scripts*, heurísticas y dominios multinivel. De esta forma, se desliga la complejidad del comportamiento del *bot* del código programado a los diferentes dominios resultando en una simplificación del código que además no quedará ligado a la resolución de un problema particular, ya que será posible reutilizarlo para generar otros *bot* capaces de interactuar con otros objetivos, juegos...

Se espera también que la aplicación de la planificación ofrezca una aproximación que induzca a un jugador humano a pensar que está jugando contra otro, es decir, más real que los *scripts* debido a la capacidad de la misma para realizar contra-estrategias diferentes, en particular si se incluyen módulos de aprendizaje automático y heurísticas.

6.1.2. Conclusiones a los objetivos específicos del proyecto

Para lograr este objetivo también se determinaron unos objetivos específicos que la aplicación debería cumplir cuyas conclusiones se exponen a continuación:

1. Comunicación: referente a la comunicación, se incluye el uso de PELEA y la comunicación con la misma. Para ello, se realizaron unos pequeños cambios dentro de la clase *Execution* de PELEA para que esta hiciese uso del servidor programado del sistema. Este servidor es el encargado de comunicarse con el cliente creado que a su vez se comunica con el juego, por lo que el primero de los objetivos queda completo.

2. Paralelismo: tanto el servidor como el cliente operan en sus propios hilos por lo que se logra el segundo objetivo específico identificado, el paralelismo de las comunicaciones con el juego dentro del servidor y el cliente.

3. Interpretación de comandos: este objetivo ha sido cumplido puesto que el sistema es capaz de reconocer los comandos que el usuario introduce en cualquiera de las dos consolas si dichos comandos han sido programados.

4. Dominio: se ha cumplido este objetivo al proporcionarse un dominio capaz de ser utilizado por el planificador con un estado enviado del cliente y que genera planes de longitud elevada.

5. Traducción y gestión del estado: se creó una representación intermedia del estado del juego con el objetivo de una gestión rápida y automática de la traducción del estado. Con esta representación intermedia se agiliza el proceso de generación del PDDL con el estado.

6. Implementación de las acciones: tras disponer de la infraestructura de comunicación, se programó un conjunto comportamientos en acciones generales que permiten la ejecución de planes complicados.

7. Traducción e interpretación de las acciones: las acciones son identificadas como tal y generadas, en caso de ser programadas como OO, o bien directamente ejecutadas, si se hace uso de la retro-llamada.

8. Ejecución y control del plan: la ejecución y control de las acciones recibidas en el plan, se ha cumplido ya que el sistema es capaz de interactuar con el juego y producir efectos visibles tal y como se definan en las acciones y el control de la ejecución de las mismas puede programarse dentro de cada acción individualmente.

9. Reutilización: en cuanto a las acciones básicas programadas, el comportamiento de las mismas se ha abstraído de tal forma que ha sido posible reutilizar parte de las mismas para ser capaces de englobar el comportamiento de varias acciones cuyo comportamiento es similar.

A modo de resumen, se concluye que el proyecto ha cumplido con los objetivos específicos determinados para el mismo.

6.1.3. Mejoras introducidas

En el proyecto se han logrado incluir comportamientos no contemplados inicialmente en los objetivos comentados con anterioridad y que proveen al sistema de una mayor versatilidad para posibles usos futuros. Estas ampliaciones a los objetivos son:

1. Comunicación: no solo se permite la comunicación entre ambas partes, si no que esta comunicación se realiza quedando almacenada la cadena de texto enviada hasta que una de las partes es capaz de analizarla.

2. Paralelismo: se ha aplicado también paralelismo para las acciones introducidas por el usuario desde el servidor, de este modo el servidor puede realizar su funcionamiento normal sin retardos.

3. Interpretación de comandos: se han ampliado las posibilidades para la ejecución de comandos del usuario, distinguiéndose entre privados del servidor, públicos del servidor, privados del cliente y públicos del cliente.

4. Dominio: se han utilizado funciones para el mejor control del juego y se ha generado un segundo dominio similar pero capaz de interactuar con las limitaciones que se han encontrado en la versión de PELEA.

5. Traducción y gestión del estado: el sistema es capaz de almacenar predicados básicos, también permite el almacenamiento de funciones pudiendo operar con ellas de forma automática. Además, el estado es capaz de almacenar objetos que no son dependientes del objeto con el cual opera el *bot*, sino que permite el almacenamiento y servicio de los objetos y de cualquier tipo de estructura definida por el usuario a las acciones de forma automática siguiendo la declaración de la acción a ejecutar enviada por el servidor.

   Finalmente, el estado es capaz de ser actualizado bajo demanda o de forma manual.

6. Implementación de las acciones: en las acciones se introdujo la marca de ejecución correcta, error o re-planificación para que las futuras acciones programadas pudieran hacer uso de las mismas.

7. Ejecución y control del plan: en cuanto a la ejecución de acciones, se ha ampliado en el sistema que pueda realizarse de dos maneras, controladas por el cliente o controladas por PELEA además de permitir la ejecución en paralelo de estas. Además, se ha ampliado el servidor para ser capaz de realizar una planificación limitada.

8. Reutilización: se ha desligado el control e interacción con PELEA de la definición de las acciones, la ejecución de las mismas y la obtención de datos por parte de los sensores del juego StarCraft pudiendo ser utilizado este código en futuras líneas de desarrollo.

6.2. Problemas encontrados

A continuación se realiza una descripción de los diferentes problemas y dificultades que se han encontrado a la hora de desarrollar este proyecto.

1. La falta de experiencia en el lenguaje de programación: si bien se contaba con formación en el lenguaje C, la forma de programación de C++ es diferente, por lo que se tuvo que investigar sobre el mismo.

2. La plataforma y el sistema operativo: no se había desarrollado código dependiente del sistema operativo Windows hasta el momento, tampoco se tenía experiencia en la programación de librerías dinámicas (DLL). Además, no se conocía la plataforma de desarrollo de Windows recomendada por BWAPI, Visual C++ Express.

3. Depuración de código: la depuración de código solo ha sido posible realizarse por el método de ensayo y error, debido a que es necesaria su ejecución en StarCraft para obtener resultados fiables. Esto ha complicado la depuración debido a que si un error se producía, el juego se cerraba sin dar más información.

4. Limitación de recursos: en un principio se utilizó la versión de PELEA que opera con Apache. Sin embargo, esta versión consumía muchos recursos del sistema por lo que el funcionamiento y desarrollo se veían fuertemente ralentizados.

5. Errores y limitaciones de la versión reducida de PELEA: PELEA es un proyecto en desarrollo, y como tal se encontraron fallos en su funcionamiento. Estos fallos condujeron a la ralentización del desarrollo de la plataforma.

   Además, el intérprete de PDDL de PELEA se encuentra en una versión limitada que no es capaz de identificar la gramática completa del lenguaje, por lo que ciertas características que se asumieron para el funcionamiento sin la herramienta no funcionaban.

6. En el funcionamiento de BWAPI ciertas características, como la gestión interna que hace para la creación de unidades, no es intuitiva por lo que fue necesario adaptar el código de las acciones generadas a la forma de trabajo de esta herramienta.

6.3. Trabajos futuros

El proyecto ha generado una arquitectura capaz de interactuar con el juego StarCraft y PELEA ofreciendo una forma sencilla para la creación de *bots*. Además, el diseño modular que se ha seguido y la independencia de cada uno de sus módulos ofrecen una gran versatilidad para su ampliación o reutilización en otros proyectos. A continuación se presentan posibilidades de mejoras a cada una de las partes, el sistema, la representación, la estrategia y PELEA.

6.3.1. Mejoras del sistema

Entre las mejoras que se pueden realizar al sistema existente se destacan las siguientes:

- El control puede ser mejorado para favorecer la ejecución de acciones en paralelo con mejores restricciones.

- El código del subsistema que opera con StarCraft, puede reutilizarse para aplicarlo a cualquier otro *bot* que opere sobre Windows teniendo en cuenta el modo de actuación actual, basado en *frames* y en el modelo de robótica sensar/planificar/actuar. De esta forma una aplicación directa correspondería con otros juegos de estrategia en tiempo real.

- Este código puede operar de forma independiente del servidor, por lo que si otro servidor es capaz de realizar la misma funcionalidad que el presentado en el proyecto, el cliente sería capaz de ejecutar las acciones proporcionadas. De la misma forma, si el servidor es utilizado por otro cliente, este sería capaz de operar con él. Además, el código programado para el mismo está diseñado para ser funcional independientemente de la plataforma en el cuál se ejecute.

- El sistema se ha diseñado para ser dependiente de Windows XP, por lo que otra posible mejora sería crear una nueva versión del sistema de control que permitiría la ejecución en diferentes plataformas.

- En la arquitectura del sistema se puede incluir otras características como la posibilidad de mantener diferentes estados sobre el mismo juego, sobre el mismo o diferente dominio, favoreciendo así la posibilidad de planificación multinivel.

- En el sistema se contempla la ejecución de acciones en el mismo hilo de ejecución del juego, pudiéndose producir retardo en el juego si estos comportamientos son complejos o numerosos. Por ello, se podría ampliar la arquitectura para ofrecer una ejecución en paralelo tanto de todas las acciones como de cada acción individualmente.

6.3.2. Mejoras en la representación de la información

En lo referente a la representación del dominio se identifican las siguientes posibles mejoras:

- Se podría hacer uso de varios dominios simplificados y creados en función del objetivo específico a resolver. De esta forma tareas que son independientes del resto pueden ser planificadas con una mayor rapidez y sin penalizar el tiempo de planificación de tareas de mayor nivel de abstracción.

- Se podría operar dentro del servidor con el fichero del dominio, modificándolo si fuera necesario en tiempo de ejecución. Esta mejora se podría combinar con la ampliación de diferentes estados formados para esos dominios que permitan la representación multinivel.

- También se podría mejorar la gestión de objetivos, tanto en el servidor como en el cliente. En el cliente se controlan de forma fija al estar programados dentro del código del bot de forma estática básica. Este comportamiento podría ampliarse para ofrecer la posibilidad de añadir un objetivo o eliminarlo durante el transcurso del juego según fuera necesario.

- En cuanto a la forma de dar de alta predicados dentro de las acciones o el sistema, se podría mejorar su implementación actual para permitir una automatización del proceso que permita realizar los cambios directamente desde una cadena de texto PDDL.

- Sobre los objetos se pueden añadir otros abstractos para gestionar las unidades de forma más eficiente que la que ofrece el juego. Una posible ampliación podría corresponderse con la gestión de las fuerzas de ataque, haciendo clasificación de las mismas en función de las unidades que contenga y extrayendo valores como su fuerza ofensiva o defensiva, sus puntos fuertes o débiles… Esta información podría ser complementada con otra extraída de los sensores tales como los datos de las unidades enemigas conocidas para poder generar dominios complejos que ejecutaran contra estrategias.

6.3.3. Mejoras en la estrategia

Sobre las mejoras en la estrategia aplicada a las acciones creadas se han seleccionado las siguientes:

- En primer lugar, las acciones básicas definidas se pueden depurar incluyendo distintas aproximaciones que contemplen diferentes situaciones y nuevos controles para los posibles eventos que surjan en el juego durante su ejecución o poder cancelarlas durante su ejecución.

- Mejorando el script de emplazamiento de edificios, se podría aprovechar mejor el terreno de cada base, evitando que se puedan crear murallas de edificios que impidan o empeoren el movimiento de las unidades, en particular de los trabajadores. Además, se podría determinar qué edificios deberán ser construidos en que base en función del tamaño de la misma y de la orografía terreno.

- En cuanto a la creación de unidades, su script se puede mejorar añadiendo ciertas variaciones de forma que se permitan que varios edificios del mismo tipo pudiesen generar unidades.

- Ambos scripts, construcción y entrenamiento de unidades, pueden ser especializados para cada raza del juego. De la misma forma, se podría generar dominios y funciones capaces de ofrecer un comportamiento más ajustado a la forma de actuar de cada raza.
- Incluyendo otras acciones, tales como formaciones de combate o control individual de unidades, se podría hacer uso de la planificación para controlar las batallas o bien definir de forma programada el comportamiento para ofrecer mejores tiempos de respuesta.

- Es necesaria la inclusión de otras acciones para el control completo de una raza. Tales acciones servirían para habilitar investigaciones o reemplazamiento de edificios para la raza Terran. Estas acciones pueden ser aprendidas mediante el uso del aprendizaje automático.

- Se puede usar la arquitectura ya existente para generar acciones de ejecución parcial o condicional ya que el sistema en su implementación actual lo permite. Esto puede ser de utilidad, por ejemplo, cuando un constructor ha sido eliminado sin haber terminado la construcción. En este escenario, se podría terminar el actuador dando de alta un predicado que indique que el edificio existe pero no está finalizado.

- Finalmente, el sistema generado en este proyecto opera utilizando la planificación. Sin embargo, es posible la incluir directamente comportamientos reactivos en repuesta a los eventos de los sensores. Para este fin se podría utilizar la versión de PELEA con dos niveles de abstracción que permita que el sistema reactivo pueda gestionar errores.

6.3.4. Mejoras en el sistema de control deliberativo (PELEA)

Las mejoras que se han estimado oportunas mencionar para PELEA son las siguientes:

- PELEA ofrece la posibilidad de uso de diferentes planificadores incluso en la versión reducida utilizada en este proyecto. Usar distintos planificadores puede resultar en comportamientos diferentes según funcionen sus heurísticas.

- Se pueden añadir módulos contemplados en su arquitectura y hacer uso de otras técnicas dentro de la planificación tales como la planificación multinivel, el uso de heurísticas o de gestión del tiempo. Con estos módulos se podría generar un *bot* con comportamientos diferentes en cada partida que fuera capaz de rivalizar con jugadores humanos.

- Si los objetivos son determinados por el servidor, este control se podría realizar tratando directamente sobre el estado recibido del juego o bien usando el módulo “goals and metrics” de PELEA.

- El intérprete de ficheros PDDL a XPPDL se podría mejorar para incluir las operaciones con variables.

Capítulo 7: Anexos

7.1. Manual de instalación

En este apartado se describe la manera de instalar los elementos necesarios tanto para desarrollar un *bot* como para probarlo en StarCraft. Es importante que dicho juego se encuentre instalado ya en la máquina dónde se probará el *bot*, aunque esta máquina puede ser independiente del desarrollo del *bot* puesto que para ello se necesita solo Visual Studio y el módulo de ejemplo o moduloIA.

7.1.1. Requisitos

- Requisitos de StarCraft

Requisito	Valor
Sistema operativo	Microsoft Windows 95
Procesador (CPU)	90 MHz Pentium o equivalente
Memoria RAM	16 MB
Tarjeta gráfica (GPU)	SVGA que soporte DirectDraw a 640x480 píxeles de resolución y 256 colores
Espacio en disco (HDD)	80 MB libres

Tabla 98: Requisitos de StarCraft

- Requisitos PELEA

La versión reducida de PELEA requiere de Java JRE 1.7 ó superior.

- Requisitos Metric-FF

La versión que usa el sistema requiere del compilador GNU GCC.

- Requisitos mínimos sistema y del entorno

Requisito	Valor
Sistema operativo	Windows Server 2003; Windows Server 2008; Windows Vista; Windows XP
Procesador (CPU)	1.6 GHz, 2,4GHz para Vista.
Memoria RAM	384 MB. 768 MB para Vista.
Tarjeta gráfica (GPU)	Pantalla de 1024x768 píxeles de resolución.
Espacio en disco (HDD)	800 MB espacio en disco.

Tabla 99: Requisitos del sistema

7.1.2. Instalación

Los pasos necesarios para una instalación completa son:

1. Si no se tiene instalado Starcraft, ejecutar "*Setup.exe*" o desde el "autorun.exe" contenido en la raíz del CD del juego. Instalar en la ruta deseada y esperar. La ruta sugerida es la que la instalación muestra por defecto. Terminada la instalación de StarCraft, se deberá seguir un paso análogo al anterior con su expansión, Broodwar.

2. Una vez instalado la expansión será necesario ejecutar el juego y comprobar en la pantalla de bienvenida la versión del juego, se encuentra en la esquina inferior derecha. Si la versión difiere de 1.16.1 será necesario dirigirse a la página de Blizzard Entertaiment® y descargar el parche para esa versión. La información sobre la versión se indica en la figura 58.

Figura 58: Versión de StarCraft

Es importante tener en cuenta que no todas las versiones de BWAPI funcionan con todas las versiones de Chaoslauncher, ni todas las versiones de Chaoslauncher con las de Starcraft. Para saber compatibilidades se recomienda visitar el hilo de Chaoslauncher (el mismo que provee los enlaces de descargas) para buscar información al respecto.

3. Descargar el fichero comprimido con la versión de ChaosLauncher compatible desde el hilo.

4. Extraer el zip en una carpeta del sistema.

5. Copiar los contenidos de Chaoslauncher/Release/ a la carpeta de Chaos Launcher.

6. Copiar los contenidos de Starcraft/ a la carpeta del juego Starcraft.
7. Copiar los contenidos de WINDOWS/ a C:\WINDOWS o C:\WINNT según la versión de Windows

8. Abrir ExampleProjects.sln en VC++ 2008 SP1 o VC++ 2008 Express Edition y compilar el proyecto como RELEASE.

Para comprobar la instalación de ChaosLauncher con el *Example AI Module*:

8.1 Copiar Release/ExampleAIModule.dll a <carpeta Starcraft>/bwapi-data/AI/ExampleAIModule.dll

8.2 Abrir <carpeta Starcraft>/bwapi-data/bwapi.ini y cambiar (si no está cambiado). ai_dll a ExampleAIMdoule.dll

8.3 Ejecutar Chaoslauncher con BWAPI Injector seleccionado. También se puede usar el módulo para la vista en ventana en vez de pantalla completa. Estas opciones se muestran en la figura 59.

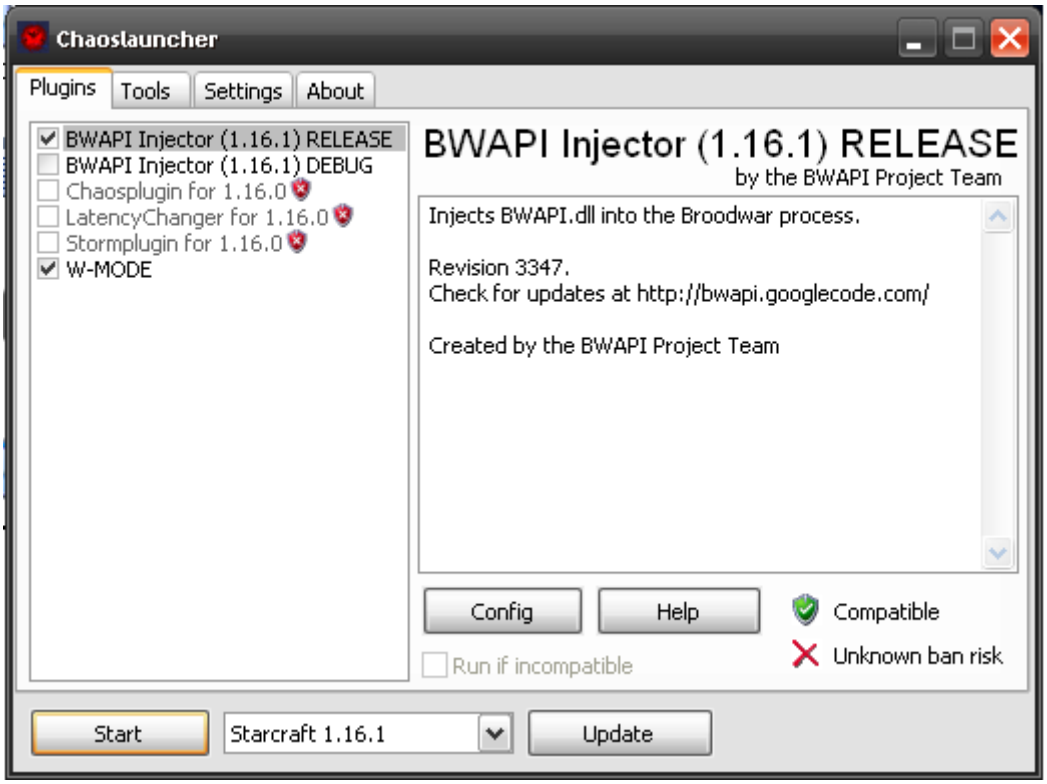

Figura 59: Configuración de lanzamiento de ChaosLauncher

8.4 Ejecutar Starcraft: Broodwar y crear una partida.

9. Descomprimir el fichero que contiene PELEA en una carpeta del sistema.

10. Descargar el planificador Metric-ff para el sistema operativo en que se encuentre PELEA.

11. Descomprimir del fichero “Metric-FF.tgz” en la carpeta deseada.

12. Navegar hasta la carpeta descomprimida desde una consola y ejecutar el comando “*make*” que generará los archivos necesarios y creará el ejecutable “ff” del planificador.

Nótese que para el funcionamiento del servidor básico proporcionado, la carpeta del planificador debe encontrarse dentro de la carpeta base “PELEA” y los ficheros de dominio deben ser generados dentro de la carpeta “Metric-FF” que contenga el planificador.

7.1.3. Configuración y arranque del sistema

En este apartado se van a describir los valores que será necesarios sustituir en los ficheros de configuración del sistema para poder ejecutarlo así como lanzar el servidor. Para ello se deberán seguir los siguientes pasos:

El fichero de configuración de PELEA, “configuration.XML”, está situado en la carpeta base. ”PELEA”. Su contenido se muestra en la figura 60.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<define xmlns="http://www.pelea.org/xPddl">
  <configuration>
        <term value="127.0.0.1" name="IP"/>
        <term value="4004" name="PORT"/>
    <term value="NOT" name="TEMPORAL"/>
    <initialize>
         <term value="/home/javier/Escritorio/PELEA/planificadores/Metric-FF/StarCraft.pddl"
name="DOMAIN"/>
        <term value="/home/javier/Escritorio/PELEA/planificadores/Metric-FF/estado.pddl"
name="PROBLEM"/>
    </initialize>
        <decissionSupport>
                <term value="org.peleame.nodes.decissionSupport" name="CLASS"/>
                <term value="1" name="NUMPLANNERS"/>
                <term value="/home/javier/Escritorio/PELEA/" name="TEMP_DIR"/>
                <term value="/home/javier/Escritorio/PELEA/planificadores/Metric-FF/"
name="PLANNER_DIR_1"/>
                <term value="org.pelea.planners.MetricFF" name="PLANNER_CLASS_1"/>
                <term value="1" name="PLANNER_MODE_1"/>
                <term value="false" name="DS_ALLWAYS_REPLANN"/>
                <term value="true" name="DELETE_TEMP_FILES" />
        </decissionSupport>
        <execution>
                <term value="org.peleame.nodes.execution" name="CLASS"/>
                <term value="127.0.0.1" name="IP"/>
                <term value="2001" name="PORT"/>
                <term value="true" name="EXECUTION_MODE" />
        </execution>
        <monitoring>
                <term value="org.peleame.nodes.monitoring" name="CLASS"/>
                <term value="true" name="VALIDATE_STATE"/>
                <term value="false" name="SENSORS_DYNAMIC"/>
        </monitoring>
        <nodes>
                <term value="1" name="DECISSIONSUPPORT"/>
                <term value="1" name="EXECUTION"/>
                <term value="0" name="GOALSMETRICS"/>
                <term value="0" name="HIGHLEVELREPLANER"/>
                <term value="0" name="LOWLEVELPLANNER"/>
                <term value="0" name="LOWTOHIGH"/>
                <term value="1" name="MONITORING"/>
                <term value="0" name="LEARNING"/>
        </nodes>
        <debug>
                <term value="ON" name="ACTIVE"/>
        </debug>
  </configuration>
</define>
```

Figura 60: Fichero de configuración de PELEA

Los valores de sus términos se recogen en la siguiente tabla:

Etiqueta	Valor	Significado
Define xmlns	http://www.pelea.org/xPddl	Define el espacio de nombres de XPDDL.
Configuration	IP	Establece la dirección IP del servidor de PELEA, local.
	PORT	Establece el puerto del servidor, en este caso 4004.
	TEMPORAL	Indica que no se va a usar fichero temporal
initialize	DOMAIN	Indica la ruta **absoluta** al fichero de dominio.
	PROBLEM	Indica la ruta **absoluta** al fichero de problema.
decissionSupport	CLASS	Define la clase que contiene el comportamiento del módulo *DecisionSupport*.
	NUMPLANNERS	Indica la cantidad de planificadores.
	TEMP_DIR	Establece la ruta absoluta para el directorio temporal.
	PLANNER_DIR_1	Establece la ruta absoluta del planificador
	PLANNER_CLASS_1	Indica a PELEA que tipo de planificador se usa.
	PLANNER_MODE_1	Indica a PELEA el modo de funcionamiento del planificador.
	DS_ALLWAYS_REPLANN	Establece que siempre se re-planifique a falso.
	DELETE_TEMP_FILES	Indica a PELEA que elimine los ficheros temporales.
execution	CLASS	Indica a PELEA la clase que contiene el comportamiento del módulo de ejecución.
	IP	Establece la dirección IP para el módulo.
	PORT	Establece el Puerto para el módulo.
	EXECUTION_MODE	Establece el valor del modo de ejecución a verdadero.
monitoring	CLASS	Indica a PELEA la clase que contiene el comportamiento del módulo.

Etiqueta	Valor	Significado
	VALIDATE_STATE	Establece la validación de estado a verdadero.
	SENSORS_DYNAMIC	Establece el tipo de sensores como dinámicos a falso.
nodes	DECISSIONSUPPORT	Indica que se va a usar el módulo *decision support*.
	EXECUTION	Indica que se va a usar el módulo *execution*.
	GOALSMETRICS	Indica que no se va a usar el módulo *goals metrics*.
	HIGHLEVELREPLANER	Indica que se no va a usar el módulo *high level replaner*.
	LOWLEVELPLANNER	Indica que se no va a usar el módulo *low level planner*
	LOWTOHIGH	Indica que se no va a usar el módulo low to *high.*
	MONITORING	Indica que se va a usar el módulo *monitoring*
	LEARNING	Indica que se no va a usar el módulo *learning.*
debug	ACTIVE	Establece el envío de mensajes de depuración a activado.

Tabla 100: Fichero de configuración de PELEA

1.1. En primer lugar, los valores que se deberán ajustar para utilizar el sistema tal y como se entrega dentro del fichero de configuración de PELEA son:

1.1.1. Dentro de la etiqueta “initialize”: localizar el nodo cuyo nombre es “DOMAIN” y sustituir la ruta completa por el fichero donde se encuentra el dominio definido. Por defecto, apunta dentro de la carpeta de “planificadores/Metric-FF”, lugar donde se encuentra el planificador compilado. Sin embargo, esta ruta es absoluta y será necesario actualizarla. En esta misma etiqueta, localizar el nodo con nombre *PROBLEM* y realizar el mismo paso que para *DOMAIN*.

1.1.2. Dentro de la etiqueta *decisionSupport*: localizar la etiqueta cuyo nombre es PLANNER_DIR_1 y establecer su valor igual que con *DOMAIN* y *PROBLEM*.

2. Para configurar el cliente, será necesario acceder a la carpeta donde la DLL se encuentra, “<carpeta Starcraft>/bwapi-data/AI/” y crear un fichero “Configuracion.xml” con el siguiente contenido mostrado en la figura 61.

```
ip=192.168.1.33

puerto=4004

control=1

paralelo=0

repetidas=1
```

Figura 61: Contenido del fichero de configuración del cliente

Las opciones son las siguientes:

- IP: en la etiqueta “IP” de dicho fichero se incluirá la dirección IP actual de la máquina en la cual el servidor se está ejecutando.
- Puerto: si se realizan modificaciones en la clase Servidor para establecer otro puerto al indicado por defecto, se deberá especificar el mismo en este fichero en la etiqueta “puerto”. En caso contrario se indicará si se desea el puerto por defecto “4004”.
- Control: indica si el control de la aplicación es externo, 1, o si es del sistema, 0.
- Paralelo: habilita la opción de ejecución de acciones en paralelo.
- Repetidas: habilita la posibilidad de recepción de acciones con el mismo nombre y que por tanto serán ejecutadas.

3. El fichero de BWAPI.ini, proporciona además la posibilidad de configurar que varios *bots* sean enfrentados. Se muestra el fichero en su configuración por defecto en la figura 62.

```
; Path for single-instance AI

ai_dll = bwapi-data\AI\ModuloIA.dll

; The following line is for BWAPI developers only

;ai_dll = bwapi-data\AI\TestAIModule.dll


; Paths for multi-instance AI, can add as many entries as needed

ai_dll_1 = bwapi-data\AI\AI_01.dll

ai_dll_2 = bwapi-data\AI\AI_02.dll

ai_dll_3 = bwapi-data\AI\AI_03.dll

ai_dll_4 = bwapi-data\AI\AI_04.dll
```

Figura 62: Configuración por defectp de BWAPI.ini

Como puede observarse, dispone de dos apartados diferenciados para el desarrollador de *bots*, el cargador para instancia única, "ai_dll" y los diferentes cargadores para los diferentes bots que compitan en multijugador categorizados como ai_dll_x dónde x indica el número del *bot*.

Para la ejecución del sistema es requisito que el servidor se encuentre en ejecución con anterioridad al lanzamiento del juego. Si se utiliza PELEA, se ha instalado dentro de un entorno Linux y se dispone del programa "Konsole", el lanzamiento de PELEA con el Servidor se reduce a ejecutar el fichero "launcher.sh" que se encuentra en la carpeta base, PELEA. En caso contrario, será necesario abrir tres terminales dentro de la carpeta base y ejecutar los siguientes comandos por este orden:

1. java -jar ./dist/pelea1Level.jar -c ./configuration.xml -n GM -t 3 -m 2
   - Arranca el módulo "monitoring" como servidor RMI de PELEA.

2. java -jar ./dist/pelea1Level.jar -c ./configuration.xml -n DS1 -t 1 -m 2
   - Lanza el módulo "*decision support*" como cliente RMI.

3. java -jar ./dist/pelea1Level.jar -c ./configuration.xml -n Sistema -t 2 -m 2
   - Lanza el módulo "*execution*" como cliente RMI.
   - Este módulo proporciona la consola de servidor del sistema.

La sintaxis que siguen estos comandos es:

- -c: indica al módulo el fichero de configuración en XML.
- -n: establece el nombre único para el nodo.
- -t: es un valor numérico que define el tipo de nodo:
  - 1: *Decision Support.*
  - 2: *Execution.*
  - 3: *Goals and Metrics.*
  - 5: *Low Level Planner.*
  - 6: *Low to High.*
  - 7: *Monitorning.*
  - 8: *Learning.*
- -m: modo de ejecución:
  - 1: servidor.
  - 2: cliente.
  - 3: individual (experimental).

En caso de desear lanzar el servidor con funcionamiento limitado sin PELEA, se recomienda abrir el proyecto dentro de un entorno de desarrollo NetBeans, localizar la clase "ServerWrapper.ServidorBasico" y ejecutarla.

Para el lanzamiento del cliente, será necesaria la ejecución de ChaosLauncher del mismo modo indicado para la prueba de la instalación del módulo de ejemplo de BWAPI; es decir, seleccionar las opciones en ChaosLauncher mostradas en la figura 63.

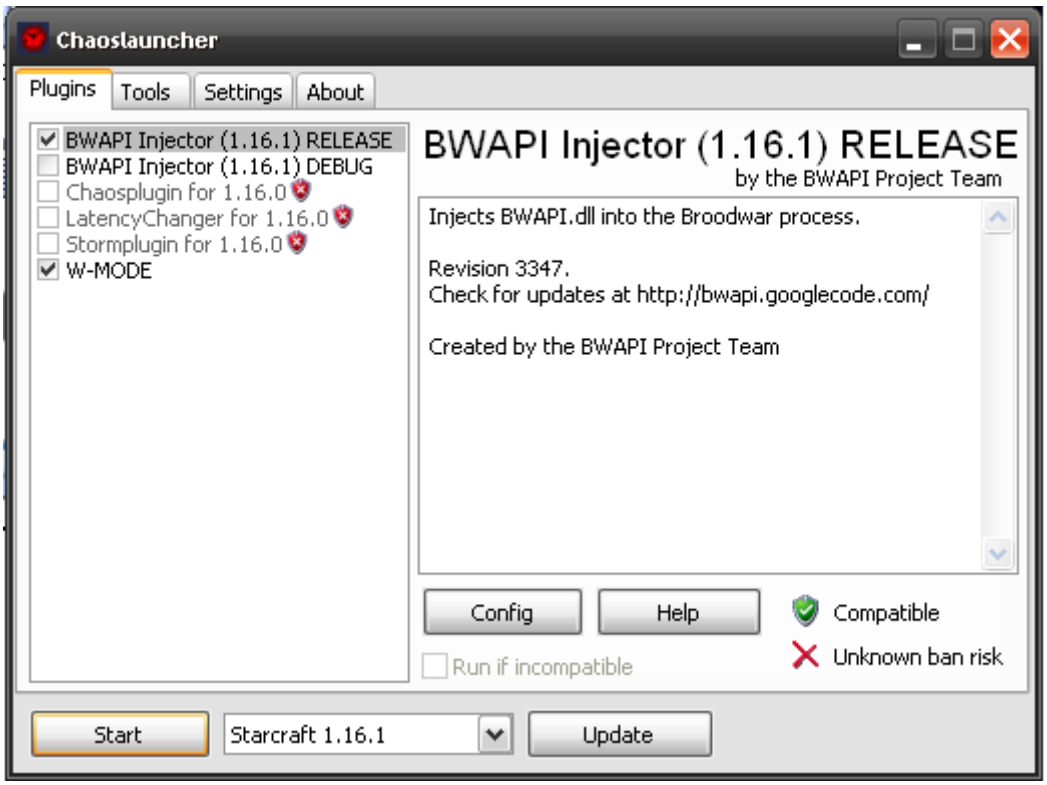

Figura 63: Opciones de lanzamiento

Presionar “Start”, y crear una partida nueva dentro del juego.

7.1.4. Nota final

Cada vez que se realicen cambios en el comportamiento del *bot* en el cliente, es necesario recompilar y copiar el DLL a la carpeta IA además de relanzar el juego. En caso de instalarse el cliente dentro de una máquina virtual, será necesario configurar las opciones de red de la misma para seleccionar la opción “Adaptador puente”.

7.2. Manual de usuario

En este documento se detallan las instrucciones para un usuario desarrollador que quiera crear un *bot* que utilice el sistema presentado en este proyecto.

7.2.1. Usuarios del sistema

El sistema diferencia entre los siguientes tipos de usuarios que pueden estar supervisando y controlando el mismo simultáneamente pero que pueden realizar distintas operaciones:

- Usuario del cliente: este usuario es capaz de interactuar con el sistema a través del envío de mensajes desde la consola que ofrece StarCraft, concretamente a través del evento recogido por la función *onSendText* dentro de la clase Sensores. Dentro de esta función existen programados unos comportamientos privados que sólo el usuario del cliente es capaz de ejecutar. Si no se encuentra el comando dentro de esta función se llama a ordenJugador dentro de núcleo. Esta función contiene los comandos públicos que tanto el usuario cliente como servidor son capaces de ejecutar. Este usuario debe disponer de los conocimientos relativos a los comandos implementados en el sistema.

- Usuario del servidor: el usuario del servidor posee la misma aproximación que el usuario cliente en cuanto a comandos públicos y privados. En el caso de comandos privados, el comportamiento se encuentra dentro de la clase IOPantalla en el método "ejecutarOrden". De no encontrarse un comando coincidente con el introducido, el método "tratarMensaje" dentro de la clase "Servidor" es invocado. Este método contiene los comandos públicos que el cliente puede solicitar al servidor. Este usuario debe disponer de los conocimientos relativos a los comandos implementados en el sistema.

- Usuario desarrollador: el usuario desarrollador es el encargado de programar el *bot* que el sistema ejecutará así como de gestionar la configuración que el sistema utilizará durante su ejecución. Este usuario debe de disponer de conocimientos del lenguaje de programación C++, así como de la estructura del sistema y sus ficheros de configuración. Deberá conocer además el funcionamiento del juego "StarCraft", sus unidades, características, habilidades, tecnologías y mecánicas para cada raza. También será necesario que el usuario desarrollador esté familiarizado con las opciones que BWAPI ofrece para el control de las unidades y la obtención de información del juego.

7.2.2. Comandos públicos y privados

El sistema permite la ejecución de comandos durante la ejecución del sistema y que son tratados de forma diferente en función de dónde son recibidos, si son recibidos directamente sobre una de las partes son considerados comandos privados de esa parte en caso contrario, públicos. La extensión de la cantidad de dichos comandos que el sistema puede reconocer

durante su ejecución puede dotar al sistema modificado de una mayor capacidad de respuesta y de adaptabilidad. Los comandos para cada subsistema son:

- Comunicación StarCraft: si se desean modificar los comandos privados del cliente, habrá que introducir los nuevos comportamientos dentro de la función "onSendText" perteneciente a la clase "Sensores". Para los comandos públicos, por el contrario, se deberá incluir dentro de la función "ordenJugador" de la clase Núcleo. Los comandos del mismo son:

  - Privados: los comandos considerados privados son los programados dentro del sistema, los pertenecientes al juego no son tratados. Estos mensajes en inglés son proporcionados por BWAPI.

    - /show bullets: indica a BWAPI que dibuje las trayectorias de los disparos.

    - /show players: muestra los jugadores de la partida.

    - /show forces: muestra las distintas fuerzas o alianzas en la partida.

    - /show visibility: dibuja marcas sobre la visibilidad del jugador en el mapa.

    - /analyze: analiza el terreno en busca de regiones y cuellos de botella. En la figura 64 se muestra un ejemplo del resultado del análisis.

Figura 64: Análisis de terreno

- Públicos
  - (Acción [parametro]*): dentro de los comandos públicos se dispone de uno previo a los comandos públicos, el tratamiento de una acción recibida en PDDL, su decodificación, inicialización y apéndice a plan almacenado. Se ejecuta en el momento en que se recibe un paréntesis como primer carácter.
  - /plan: indica que se dispone de un plan a ejecutar.
  - /leem: fuerza la lectura de mensajes del buzón de recepción.
  - /aPDDL: muestra en el juego el estado en PDDL.
  - /noplan: indica que no se dispone de plan para ejecutar.
  - /planificar: ordena al servidor una planificación.
  - /getSensors: envía al servidor la información del estado en PDDL.
  - /enviar'estado: envía el estado en PDDL y le indica que el mensaje es el estado.
  - /enviar'socket: envía una cadena de texto al servidor.
  - /mostrar'dominio: muestra el contenido del dominio por la consola del juego.
  - /conectar'socket: fuerza una conexión del socket.

- En el servidor, los comandos privados se encuentran incluidos dentro de la clase IOPantalla en el método "ejecutarOrden". Los comandos públicos, se encuentran dentro del método "tratarMensaje" de la clase "Servidor". Los comandos reconocidos en minúscula son:
  - Privados
    - Enviar: permite el envío de una cadena de texto al cliente. Se puede utilizar para ejecutar las acciones públicas del cliente.
    - Contar: cuenta los caracteres de una cadena de texto.
    - Conectar: intenta una conexión.
    - Desconectar: desconecta el servidor.
    - Setpuerto: solicita al usuario el puerto que usa el servidor. Será necesario desconectarlo si se encuentra conectado.
    - Tratarmensaje: solicita una cadena que será tratada dentro de los comandos públicos del servidor.

- Públicos: estos mensajes comienzan con la cadena “/m:”
  - Eom: indica el final de un mensaje, para su volcado desde el búffer de recepción al mensaje almacenado.
  - Estado: indica que el mensaje almacenado es un estado, y por tanto, debe ser almacenado en un fichero en la ruta del planificador.
  - OK: el resultado de la acción cuyo nombre se especifica después de los dos puntos ha sido satisfactorio.
  - Planificar: ordena una ejecución manual del planificador. Tras la ejecución, se extrae el plan, se indica si se ha obtenido plan al cliente y se envía la primera acción.

7.2.3. Creación de bots

En este apartado se procede a describir el proceso de creación de un *bot* dentro del sistema que haga uso de todas las funcionalidades implementadas. El desarrollador de *bots* podrá desarrollar su código dentro de cuatro clases especialmente preparadas para este fin.

7.2.3.1. *Definición de dominio*

Los *bots* generados deben trabajar sobre un dominio sobre el cual el *bot* pueda plasmar un estado y el planificador realizar su tarea. Este dominio es necesario que sea generado en forma de un fichero PDDL que se debe encontrar almacenado en el servidor y cuya ruta absoluta se encuentre correctamente configurada dentro del fichero de configuración de PELEA o en su defecto en la carpeta “planificadores/Metric-FF”.

En cuanto a las capacidades, el sistema admite el funcionamiento probado de funciones simples, así como de las características básicas, predicados, tipado de objetos, y de acciones parciales y/o condicionales. Estas acciones deberán estar incluidas dentro del cliente o bien como implementaciones mediante retro-llamadas, o como instanciación de un objeto. No es necesario, sin embargo, que cada acción sea implementada directamente, pudiéndose reutilizar una misma función o script haciendo que la traducción del nombre de la misma enlace con el comportamiento deseado.

Un ejemplo sencillo se muestra en la figura 65. En ella se definen los tipos de objetos correspondientes a una base, un depósito y un juego. Este dominio almacena los recursos disponibles en forma de funciones con la población actual, la máxima, los depósitos de mineral y la cantidad de trabajadores ocupados y disponibles. También define si en una base se pueden crear unidades, si se ha alcanzado el máximo de población y si se dispone de trabajadores. Con estos datos, se definen dos acciones, una para la creación de un trabajador y la otra para ordenarle la recolección de minerales.

```
(define (domain StarCraft)
(:requirements :typing :fluents)
(:types Deposito_mineral Base Juego)
(:predicates
        (Poblacion_maxima ?j - Juego)
        (Puede_crear_unidades ?b - base)
        (Trabajadores_disponibles ?b -Base))

(:functions (minerales)
            (frames)
            (poblacion_actual)
            (poblacion_maxima)
            (depositos_mineral)
            (trabajadores_mineral)
            (trabajadores_desocupados))

(:action CrearTrabajadores
:parameters (?b - Base ?j -Juego)
:precondition (and (not (Poblacion_maxima ?j))
                    (>= (minerales) 50)
                    (Puede_crear_unidades ?b)
                    (< (poblacion_actual) (poblacion_maxima)))

:effect (and (Trabajadores_disponibles ?b)
            (increase (frames) 300)
            (increase (trabajadores_desocupados) 1)
            (decrease (minerales) 50)
            (increase (poblacion_actual) 2)))



(:action RecolectarMineral
:parameters (?b - Base)
:precondition (and (Trabajadores_disponibles ?b)
                   (not  (Recoleccion_total_mineral ?b))
                    (> (trabajadores_desocupados) 0)))

:effect (and (not (Trabajadores_disponibles ?b) )
             (increase (trabajadores_mineral) (trabajadores_desocupados))
             (assign (trabajadores_desocupados) 0))))
```

Figura 65: Ejemplo de dominio

7.2.3.2. *Definición de tipos de objetos*

Para la gestión complejos, se dispone de la clase “EstructurasAbstractas”. En esta clase se almacenan las definiciones de las estructuras de los objetos que se deseen almacenar además de disponer de una función invocada en el momento en que es requerido una representación del estado: “actualizarFunciones”.

En esta función se extraerán los valores de las funciones y los predicados que se deseen dar de alta en el momento en que la representación es solicitada.

7.2.3.3. *Implementación de acciones*

Para la implementación de acciones se dispone de dos mecanismos: la orientación a objetos y la retro-llamada. Ambos comportamientos han de ser programados dentro del cliente y deben ser dados de alta en el sistema dentro de la misma clase del sistema, “Actuadores”,

Implementación de acciones por puntero a función

En cuanto a la implementación por puntero a función, el comportamiento de las funciones que reciban la llamada puede implementarse en cualquier clase, aunque los ejemplos programados dentro del sistema de ejemplo se encuentran dentro de la misma clase “Actuadores”. Para añadir un comportamiento será necesario seguir los siguientes pasos:

1. Creación de la función objetivo: en este caso la función es “wrapper_botonDerecho” siendo la declaración:

```
static std::string wrapper_botonDerecho(void* Sensores, void* objprincipal, void* params,bool activo);
```

Los parámetros que recibe la función objetivo han de ser los que se listan a continuación:

- Sensores: el puntero al objeto que representa los sensores y que contiene un acceso al juego.
- Objprincipal: el primer objeto que la acción requiere.
- Params: un vector que contiene el resto de los objetos que la acción solicitó en la declaración.
- Activo: si la acción ya ha sido activada con anterioridad.

2. Añadirla al sistema: para ello se ha de modificar el cuerpo de la función:

```
void registrarActuadores(Dominio * dominio)
```

La sintaxis para definir la acción que se corresponde con la definición en PDDL (figura 66):

(Recolectar ?t – Trabajador ?dm - Deposito_mineral)

Que produce el efecto:

(Recolectando t dm)

```
//Inicialización de la acción Recolectar ¿t - trabajador dm -
//Deposito_mineral
vector<string> tiposObjetos; //Nombres de los tipos de objetos
vector<string> pseudoNombres; // Representación reducida de los
tipos
string accion; //Nombre de la acción
vector<vector<string>> predicados; //Efectos
vector<string> predicado;

//Cabecera de la acción
accion = "Recolectar";
tiposObjetos.push_back("Trabajador");
tiposObjetos.push_back("Deposito_mineral");

pseudoNombres.push_back("t");
pseudoNombres.push_back("dm");

//Efectos de la acción
predicado.push_back("Recolectar");
predicado.push_back("t");
predicado.push_back("dm");
predicado.push_back("true");// Añadir o eliminar el predicado


predicados.push_back(predicado);

dominio->anyadirAccion(accion,   tiposObjetos,   pseudoNombres,
wrapper_botonDerecho, predicados);
```

Figura 66: Ejemplo para añadir una acción

Nótese que no se requiere incluir las precondiciones de la acción puesto que estos serán comprobados en el momento en que se solicite la ejecución de la acción, es decir, en cuanto la función enlazada objetivo es ejecutada. Con esta aproximación sólo es necesario realizar las comprobaciones de los prerrequisitos de la acción dentro de la función objetivo y, en caso de cumplirse, ejecutar la acción.

Implementación de acciones objetos

La implementación por objetos permite desarrollar acciones capaces de utilizar las ventajas que ofrece la herencia y el polimorfismo. En el sistema se ha dispuesto de tal forma que extendiendo una clase en concreto el sistema pone a la disposición de la acción cuando es ejecutada los objetos necesarios para su funcionamiento. Para crear una acción se ha de seguir los siguientes pasos:

1. Crear la clase base: Para disponer de una clase nueva que contenga todos los elementos necesarios para una acción básica se ha de heredar de la clase Actuador. Los atributos que son enlazados automáticamente y de los que se puede hacer uso son:
   - Dominio: la instancia del dominio sobre la cual el cliente opera inicialmente. Esta clase contiene también la instancia del estado actual y las funciones para operar sobre él.
   - Escritor: la instancia a la clase encargada de almacenar los datos en el registro.
   - estructurasAbstractas: la definición de estructuras contenida en la clase "EstructurasAbstractas".
   - Game: la instancia del juego sobre la cual operar. Esta instancia ha de ser de tipo "BWAPI::Game".
   - vector<void*> objetos: contiene los objetos enlazados almacenados como punteros.
   - string estado: almacenamiento del estado del actuador.
   - int saltarFrames: indica al cliente que el actuador está esperando una cantidad de tiempo determinada en el número de *frames* y hasta que este contador no llegue a 0, no se ordenará su ejecución nuevamente.
   - bool accionCancelable: definición de la acción como interrumpible.
   - bool accionEnEjecucion: estado de ejecución de la acción.

2. Instanciado del actuador: la instanciación del Actuador programado, debe ser realizada dentro de la función de la clase "Actuadores":

```
Actuador* construirActuador(string nombre, vector<string> parametros)
```

Esta función recibe el nombre del actuador a crear tal y como ha sido definido en la acción dentro del fichero de dominio. En función de este nombre, se dispone de un "*switch*" que opera sobre la longitud de dicha cadena de texto y en cuyos "*case*" ha de incluirse la instanciación de un nuevo objeto del actuador deseado que ha de ser almacenado dentro del puntero declarado al inicio de la función "`actuador`". Los argumentos del constructor de dicho actuador en caso de ser requeridos, deben ser procesados en las instrucciones previas a la instanciación de forma manual.

El enlazado con el dominio, juego y datos necesarios puede hacerse de forma manual o automática. El comportamiento automatizado se encuentra al final de dicha función mediante la llamada a otra función en la clase "Actuadores". Por eso no es necesario realizar el trabajo de enlazado si los objetos concretos sobre los que opera la acción están definidos en los parámetros que recibe la misma, es decir, los objetos que son enlazados se extraen directamente de los parámetros de la acción a ejecutar. Esta acción es recibida en PDDL desde el servidor y se corresponderá con su definición en el dominio. Dicha función es:

```
void inicializarActuador(actuador, parametros)
```

Esta función enlazará los siguientes objetos por defecto al actuador:

```
actuador->setDominio(this->dom);
actuador->setEscritor(this->escritor);
actuador->setEstructurasAbstractas(this->estructurasAbstractas);
actuador->setGame(this->Broodwar);
actuador->inicializarActuador(parametros);
```

Dichos objetos se corresponden respectivamente con:

- Dominio: La instancia del dominio sobre la cual el cliente opera inicialmente.
- Escritor: La instancia a la clase encargada de almacenar los datos en el registro.
- estructurasAbstractas: La definición de estructuras contenida en la clase "EstructurasAbstractas".
- Game: La instancia del juego sobre la cual operar. Esta instancia ha de ser de tipo "BWAPI::Game".

Estos objetos son los requisitos indispensables para el actuador genérico y básicos para el funcionamiento y depuración de los actuadores generados.

La última de las instrucciones, "inicializarActuador", realiza el enlazado automático de los objetos recibidos por parámetros con las instancias de los mismos almacenadas dentro del dominio que se le haya indicado al actuador para operar. Dichos objetos son almacenados dentro del atributo de la clase base "Actuador":

```
vector<void*> objetos;
```

Dentro de esta misma clase se definen los siguientes atributos de control:

```
//Almacenamiento del estado del actuador
string estado;
//Cantidad de frames restantes en espera para la ejecución
int saltarFrames;

//Definición de la acción como interrumpible
bool accionCancelable;
//Estado de ejecución de la acción
bool accionEnEjecucion;
```

3. Definición del comportamiento: en cuanto al comportamiento de la nueva acción, debe ser realizado sobrecargando la función:

```
virtual string ejecutarActuador();
```

Los valores reconocidos por el sistema para el valor devuelto por esta función son:

- OK: La acción ha sido ejecutada con éxito, es eliminada de la lista de acciones a ejecutar y el mensaje "OK" es enviado al servidor con el nombre de la acción.

- Re-planificar: La acción no ha podido ser ejecutada por que no se han cumplido los requisitos o se ha encontrado un problema durante la misma. Se solicita al servidor una re-planificación. No tiene efecto si el control se realiza desde el servidor.

Otro de los valores que el sistema reconoce tras la ejecución de la acción se corresponde con el valor del campo público "`mensajeError`" de la clase "EstructurasAbstractas". Este valor es el siguiente:

- Esperando: La acción se encuentra en un estado de espera no cancelable de forma que si se ha pedido una re-planificación no es descartada a la eliminación de acciones almacenadas.

Por defecto el sistema realiza una llamada a "`ejecutarActuador`" por cada fotograma transcurrido en el juego, sin embargo, si se establece un valor dentro de "`saltarFrames`", el sistema obviará automáticamente la llamada a dicha función hasta que "`saltarFrames`" valga 0. Para ello restará automáticamente un fotograma cada vez que ocurra en el juego de dicho actuador.

3.1. Modificación del estado interno: para actualizar los valores del estado, el cliente dispone de varias opciones:

- Actualización directa: se opera directamente sobre el estado almacenado del juego actualizando los predicados y funciones.

- Actualización bajo demanda: en el momento en el que el cliente recibe la orden de enviar el estado, se actualizan los valores.

En la actualización directa, estos cambios se realizan en los sensores en respuesta a un evento producido en el juego o bien en los actuadores, cuando tras la ejecución de una acción correctamente se ejecutan los efectos de la misma en el estado.

La actualización bajo demanda se efectúa en sensores dentro de la función "actualizarFunciones" si los valores de las funciones o hechos son accesibles directamente desde el juego y fácilmente calculables. Si estos valores son calculados a partir de estructuras generadas por el desarrollador, el proceso de los mismos se produce dentro de la clase EstructurasAbstractas en la función con el mismo nombre. El resto de modificaciones salvo los objetivos, se realizan a través de métodos de la clase Dominio. Para el establecimiento de los objetivos, se realizan de manera programada dentro de la clase Estado en su función "aPDDL". En esta función se definen además los requisitos del estado para el planificador y la métrica para minimizar o maximizar en basándose en las funciones del estado. El resto de funciones que la acción puede realizar dentro de su clase son:

3.1.1.Añadir nuevos objetos al estado: la función que provee Dominio para este fin es:

```
void anyadirObjeto(string tipo,string nombre,void* ptro=NULL);
```

Esta función permite añadir un objeto manualmente al dominio. Esta función no controla la nomenclatura automática de los objetos y por tanto, los nombres de los objetos no deberán seguir el formato:

- Tipo_del_objetoXX
  - Dónde XX identifica el número del objeto.

Esto es debido a que el sistema mantiene su cuenta interna sobre los objetos del dominio y podría causar colisión con otro objeto añadido del mismo tipo. Además, no se comprueba si el objeto está repetido, pudiendo producir estados con diferentes objetos enlazados por el mismo nombre, resultando en un estado erróneo al realizar la conversión a PDDL y la ocultación del segundo objeto añadido.

El parámetro "ptro" por su parte, permite añadir una referencia al objeto que se desee almacenar. Aunque no es necesario enlazar un objeto no nulo, de no hacerlo el sistema no será capaz de utilizar dicha declaración para usarla como parámetros en futuras acciones.

3.1.2. Obtención de los objetos: la función para obtener un objeto que no sea recibido por parámetros automáticamente en función de su nombre es:

```
void* getObjeto(string nombre);
```

Devuelve el puntero al objeto almacenado en el estado con el nombre especificado. Con dicho puntero es posible realizar operaciones sobre el objeto que no alteren el estado y, por tanto, no sea necesario actualizar el mismo.

3.1.3. Gestión de predicados: la gestión de los efectos de una acción puede realizarse de forma automática indicando en su instanciación en la clase actuadores que efectos produce o bien de forma manual durante la ejecución de la acción.

3.1.3.1. Alta de predicados: para dar de alta un predicado se utiliza la función:

```
string anyadirPredicado(string nombre, vector<string> params);
```

Esta función permite añadir un hecho específico que en caso de no existir previamente, será dado de alta en el estado del problema. En ella, se recibe el nombre del hecho y la lista de los nombres de los objetos que están incluidos en el predicado. Con dichos nombres se enlazarán automáticamente las instancias de los objetos registrados en el estado. En caso de no encontrarse uno de los objetos, se devolverá una cadena del tipo:

```
"El parametro " + params.at(i)+ " no existe como objeto";
```

Donde "`params.at(i)`" contiene el nombre del objeto introducido. En caso de haber sido añadido correctamente, la función devuelve una cadena vacía "".

3.1.3.2. Baja de predicados: para la eliminación de un predicado del estado se utiliza:

```
string eliminarPredicado(string nombre, vector<string> params);
```

El funcionamiento de este método es similar al anterior con la diferencia de que en este caso se elimina el predicado del estado, de existir.

3.1.3.3. Consulta de existencia: para saber si un determinado predicado existe en el estado se utiliza:

```
bool existePredicado(string nombre, vector<string> params);
```

Esta función comprueba si el predicado indicado existe dentro del estado y devuelve el resultado de dicha comprobación.

3.1.3.4. Obtener los predicados: para obtener el listado completo de predicados se utiliza:

```
vector<string> getPredicados();
```

Devuelve una lista con todos los predicados activos en el estado con el formato: [Nombre][espacio] seguido de un listado con los parámetros del mismo separados por espacios.

3.1.4.Gestión de funciones: para la gestión de las funciones se utiliza sólo la siguiente función:

```
string modificarFuncion(string nombre, vector<string> params,
vector<bool> numeros, int operacion=0);
```

En esta función se permite la modificación de los valores de las funciones. En caso de no existir la función es dada de alta. Al igual que los predicados, comprueba si los objetos existen en caso de no ser un número y los enlaza con la función. En caso de introducirse un número, será almacenado como un objeto nuevo de tipo “número”. La descripción de los parámetros es la siguiente:

- nombre: el nombre de la función a modificar.
- Params: el listado de parámetros tanto objetos como números que compongan la función.
- Números: listado de igual longitud a “params” que indica cuál de los parámetros es un número.
- Operación: el número de la operación sobre la función a efectuar. Los posibles valores se encuentran como atributos públicos constantes de la clase “Dominio” y son los siguientes:
  - INCREMENTAR: Incrementa el último de los valores de la función que ha de corresponderse con un número. En caso de no existir la función, se da de alta la misma y se establece el valor final de la misma como la cantidad a incrementar.

- DISMINUIR: Realiza el comportamiento opuesto a INCREMENTAR.
- ASIGNAR: Sustituye el valor de la función por el nuevo indicado.

4. Añadir efectos a una acción: los efectos se pueden añadir manualmente como se ha expuesto con anterioridad. Sin embargo, el sistema dispone de una función preparada para ello: `ejecutarEfectos`. Esta función se puede llamar desde cualquier parte del actuador. Su sintaxis es:

```
void anyadirEfecto(string nombre, vector<string> params,bool operacion=true);
```

Está función permite dar de alta o eliminar un predicado existente en el dominio, si *false* es especificado en la operación. Para ello, ha de especificarse el nombre del predicado así como el valor explícito que los parámetros de dicho predicado tomarán siendo este valor el nombre exacto con el que el objeto es almacenado dentro del estado. Su sobrecarga es:

```
void anyadirEfecto(string nombre, vector<string> , params,vector<bool> numeros,
                                                                    int operacion=0);
```

Añade a la anterior la posibilidad de actualizar los valores de funciones. Al igual que en la función anterior, ha de especificarse el nombre de la función y un listado de sus parámetros. Además, ha de indicarse cual de esos parámetros es un número y por tanto será ignorado mediante el vector de booleanos. La operación indica si va a consistir en un incremento, decremento o asignación. Dichos valores están definidos como constantes dentro del dominio. Para ilustrar su uso se propone el siguiente ejemplo:

```
(incrementar cuarteles 1)
```

Sería:

```
string nombre="cuarteles";
vector<string> params;
params.push_back("1");
vector<bool> numeros;
numeros.push_back(true);
int operacion=dom->INCREMENTAR;
actuador->anyadirEfecto(nombre,params,numeros,operacion);
```

5. Estado inicial: si se introduce una nueva función no contemplada es necesario que se encuentre definida desde el primer momento en el estado. Para determinar los valores de los predicados que son calculados al empezar la partida se dispone de una función en la clase sensores:

```
void establecerFuncionesIniciales();
```

   En ella es donde se recogen esos cálculos ya que es llamada al inicio de la partida y se dispone de los datos del juego directamente. Es importante tener en cuenta que todas las funciones han de estar inicializadas, aunque su valor sea 0.

7.3. Bwapi

BWAPI es un proyecto de código abierto escrito en C++ que provee de las herramientas necesarias para poder generar una DLL que, inyectada en el juego StarCraft, permita controlar el comportamiento de un jugador.

BWAPI permite realizar operaciones con unidades, enviar mensajes o consultar datos referentes al mapa tales como consultas sobre el terreno en una determinada posición, las características del mapa, estado de unidades…

A continuación se va a mostrar el diagrama de clases de BWAPI, la representación que hace de las unidades, su funcionamiento en lo referente al código que proporciona y sus otras utilidades incluidas.

7.3.1. Diagrama de clases

El diagrama de clases completo se recoge en la figura 67.

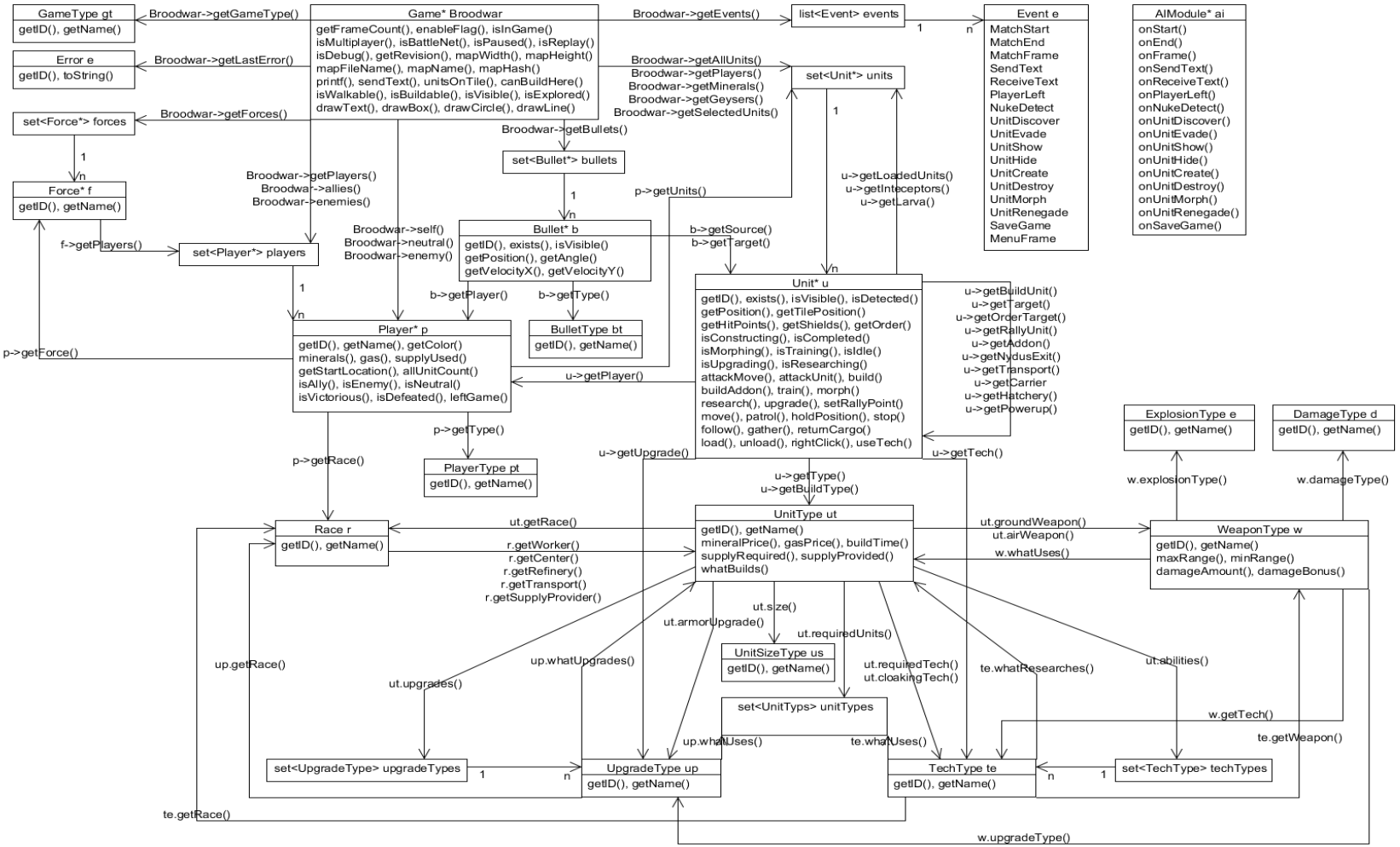

Figura 67: Diagrama completo de clases de BWAPI

Estas clases tienen el objetivo de proveer la información necesaria y los comandos adecuados para que sea posible generar un *bot*. De entre todas las clases se destacan:

- *Game*: Contiene la información relativa al juego, jugadores, fuerzas, unidades… y permite la ejecución de acciones para mover la pantalla, pausar el juego…
- *Force*: Generalmente representa los equipos que participan en la partida.
- *Player*: Contiene toda la información de cada jugador.
- *Unit*: Contiene la información relativa a cada unidad, además de permitir el control de las mimas.
- *Bullet*: Representa la información relativa a un disparo. No todos los ataques generan un objeto de esta clase.

7.3.2. Unidades en BWAPI

En BWAPI son considerados como unidades todos aquellos elementos individuales con los que se pueden interactuar, desde edificios, unidades militares, neutrales o balizas hasta los depósitos de recursos o géiseres de vespeno. Es a través de las unidades como se pueden realizar las operaciones o los movimientos en el juego.

La interacción con las unidades se realiza operando directamente sobre ellas, invocando un comando que la unidad posea y, en caso de ser necesario, indicar una posición u otra unidad sobre la cual realizar la acción seleccionada. Del mismo modo se puede consultar la información relativa a dicha unidad tanto de carácter general, puntos de resistencia restante, tipo de unidad, posición, jugador propietario… como específico: energía disponible si la dispusiera, estado de construcción, si la unidad tiene algún estado alterado…

7.3.2.1. *Posicionamiento de unidades*

La posición de las unidades en BWAPI se realiza a tres niveles:

1. Píxel: como unidad básica de medida el píxel a máxima resolución.
2. Casilla caminable: es una casilla de ocho por ocho píxeles sobre los cuales se dispone de información relativa al movimiento.
3. Casilla de construcción: está conformada por cuatro por cuatro casillas caminables, o treinta y dos por treinta y dos píxeles. Reciben esta denominación por que la información de construcción se encuentra a esa resolución.

7.3.2.2. *Almacenamiento y creación de unidades*

El sistema de almacenamiento de unidades en BWAPI se realiza de tal forma que cada unidad posee un identificador único y un puntero a dicha unidad que se mantiene durante todo el juego. Este puntero e ID es generado cuando la unidad es creada.

Sin embargo, la creación de unidades no queda notificada como cabría esperarse cuando la unidad ha sido finalmente entrenada, sino que la notificación de su creación es enviada cuando se ha creado un objeto de dicha unidad; es decir, en el mismo momento en que la unidad ha entrado en la cola de producción del edificio o ha empezado a mutar.

7.3.3. Funcionamiento

BWAPI en su módulo de ejemplo proporciona el código necesario para compilar en Visual C++ el fichero DLL que ChaosLauncher usa para inyectarlo en el juego. Este código ofrece una serie de llamadas cuando un evento se recoge en el juego. Dichos eventos están tratados como retro-llamadas (*callbacks*) a las siguientes funciones:

- *onStart:* este evento es llamado cuando la partida comienza. Su función principal es inicializar las estructuras y clases necesarias por el bot.

- *onEnd:* esta función es llamada cuando una partida finaliza indicando si el jugador ha ganado, salvo en el caso en el que la partida haya sido una repetición. En este caso siempre indica que se ha perdido.

- *onFrame:* BWAPI llama esta función en cada *frame* lógico o un pintado en pantalla del juego.

- *onSendText:* este evento recibe el texto que se ha introducido en el chat, tanto el enviado por el jugador como por otros jugadores.

- *onReceiveText:* en este evento se recibe el mensaje que otro jugador haya enviado al bot o al jugador.

- *onPlayerLeft:* se recibe una llamada en esta función cada vez que un jugador abandona el juego.

- *onNukeDetect:* esta llamada ocurre en el momento en el que se detecta un lanzamiento nuclear (habilidad única de los Terran). Si el objetivo del lanzamiento está dentro del terreno visible por el jugador, esta función recibe dicha posición.

- *onUnitDiscover: s*e activa este evento cada vez que una unidad es accesible. Si no está activada la propiedad de información completa, este evento se llama a la vez que *onUnitShow*. En caso de estar activada, se llama en el mismo momento que *onUnitCreate*.

- *onUnitEvade:* similar al evento *onUnitDiscover*, este evento se llama cuando una unidad deja de ser accesible. En caso de que la propiedad de información completa esté activada, se recibe en el mismo momento que *onUnitDestroy*. En caso contrario, a la vez que *onUnitHide.*

- *onUnitShow:* este evento es activado cada vez que una unidad se vuelve visible para el jugador.

- *onUnitHide:* se recibe esta llamada cuando una unidad deja de ser visible.

- *onUnitCreate:* BWAPI llama este evento cuando una unidad accesible ha sido creada excepto que la unidad haya sido creada mediante una transformación o en el momento de su creación la unidad es invisible.

- *onUnitDestroy:* se recibe este evento cuando una unidad accesible ha muerto o ha sido destruida del juego. Esta destrucción incluye cuando un depósito de mineral ha sido completamente recolectado.

- *onUnitMorph:* se efectúa esta llamada cuando una unidad accesible cambia su tipo.

- *onUnitRenegade:* esta llamada se recibe cuando una unidad accesible cambia de propietario.

- *onSaveGame:* se recibe una llamada en esta función cuando un jugador guarda una partida. También se recibe el nombre que el jugador dé a la partida guardada.

Además, ofrece una serie de funciones programadas que permiten mostrar información de forma gráfica en el juego. Dichas funciones son las siguientes:

- *drawStats:* muestra las unidades y la cantidad de ellas que el jugador posee en la esquina superior izquierda del juego.

- *drawBullets:* permite mostrar la trayectoria de los disparos efectuados en el juego.

- *drawVisibilityData:* dibuja un punto en el centro de cada casilla de construcción que sea visible por el jugador.

- *drawTerrainData:* si el terreno ha sido analizado, marca los depósitos de mineral y géiseres de vespeno así como la base principal del jugador. Además, genera un polígono que encuadre las diferentes regiones analizadas y una línea roja en los cuellos de botella identificados.

- *showPlayers:* muestra los nombres de los jugadores en la partida.

- *showForces:* muestra los jugadores pertenecientes a cada equipo en partida.

BWAPI indica en su módulo de ejemplo que la interacción con BWAPI debe de hacerse en el hilo por omisión en el que se ejecuta. En caso de no hacerlo, el comportamiento puede generar resultados inesperados o producir que el programa se cierre.

7.3.4. Otras utilidades

BWAPI aporta otras utilidades para el enfrentamiento simultáneo de *bots* y análisis del terreno. La primera de las utilidades corresponde a un cliente integrado que permitiría la recepción de comandos de órdenes del proceso servidor de BWAPI y su ejecución, así como el envío de los eventos que se hayan producido en el juego a dicho cliente. Sin embargo, esta aproximación implica que el proceso remoto cliente ha de conocer la implementación de BWAPI, debe de estar programado en C++ y BWAPI deberá ser configurado como servidor para poder ser aceptado.

La segunda de las posibilidades que ofrece BWAPI es la ejecución de las DLL de cada *bot* indicada en su fichero de configuración. En el desarrollo de este PFC se ha utilizado esta opción para la ejecución y pruebas del *bot* desarrollado.

En cuanto a la utilidad de análisis de terreno, BWAPI analiza el territorio visible buscando los cuellos de botella y cambios de altura o delimitadores de zona como puede ser territorio infranqueable, que permita definir regiones.

Finalmente, BWAPI muestra sus mensajes tal y como se muestra en la figura 68.

Figura 68: Mensajes de BWAPI

Donde:

1. Mensajes: En éste área se recogen tanto los mensajes que el jugador envía, como los que recibe.

2. Notificación de BWAPI: En esta área BWAPI informa de las unidades que el jugador dispone en ese momento.

Es importante destacar que la información de las unidades ocultas por la niebla de guerra no se muestra en el mini mapa.

7.4. ChaosLauncher

ChaosLauncher es un proyecto de código abierto consistente en un lanzador para StarCraft que proporciona la habilidad para inyectar *plugins* y ficheros DLL. Algunos de estos *plugins* o complementos se encuentran integrados dentro del lanzador mientras que otros son desarrollados por terceros.

A continuación se va a explicar la lista de *plugins* que van incluidos y la interfaz del programa.

7.4.1. Plugins

El listado de *plugins* incluidos en la última versión del lanzador es:

- *Chaosplugin: plugin* que guarda automáticamente las partidas jugadas, deshabilita las teclas de Windows y Bloq. Mayúsculas durante el juego, y cambiar la velocidad del ratón durante una partida.

- *LatencyChanger:* es un *plugin* que reduce el tiempo de respuesta entre jugadores en una partida a través de internet para permitir mejorar la respuesta de las unidades a las órdenes del jugador.

- *Stormplugin:* permite desactivar la barra de progreso en la repetición de una partida, desactivar los atajos de teclado para configuraciones no estándar y detectar cuándo un jugador tiene un nivel de latencia elevado.

- *W-Mode:* Permite la ejecución del juego en modo ventana, bloqueando el ratón dentro del juego durante el juego, poniendo la ventana por encima para evitar que sea minimizada y duplicar la resolución del juego sin interpolación.

- APMAlert: Muestra el nivel de acciones por minuto del jugador y avisa en caso de que una determinada frontera configurable sea sobrepasada.

- Replay Fix for 1.16.1: Soluciona un fallo por el cual la reproducción de partidas jugadas con envío de mensajes de texto durante una pausa en el juego pudiera causar que el juego se cerrase.

Entre los *plugins* de terceros se encuentra el *BroodWar API Loader,* que permite el uso de los ficheros DLL generados con BWAPI ser ejecutados dentro de StarCraft.

7.4.2. Interfaz

ChaosLauncher permite la ejecución y configuración personalizada de los *plugins* que se hayan instalado así como la instalación de estos mediante su interfaz gráfica. La interfaz gráfica principal de ChaosLauncher se muestra en la figura 69.

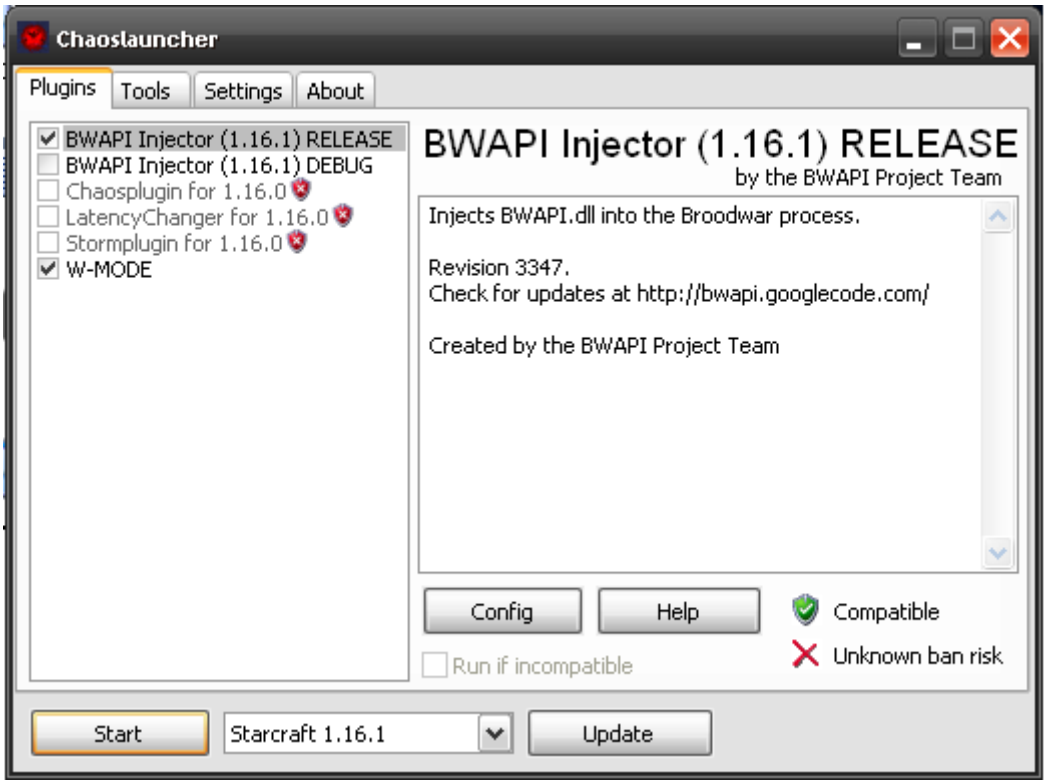

Figura 69: Pantalla principal de ChaosLauncher

En el apartado superior se puede observar un menú en forma de pestañas que permiten seleccionar el listado de los *plugins* que se ejecutarán al lanzar el juego, las herramientas (*tools*) instaladas, la configuración de ChaosLauncher y la información sobre el programa.

En la parte inferior del mismo se encuentra el botón *Start* que inicia StarCraft, un panel desplegable para la versión del juego y el botón de *update*. Que permite la actualización de Chaoslauncher.

Se va a proceder a describir cada una de las pestañas que provee esta interfaz.

7.4.2.1. *Plugins:*

La primera pestaña, *Plugins*, muestra en el recuadro de la izquierda el listado de *plugins* instalados así como aquellos marcados para la ejecución en el lanzamiento del juego. También muestra iconos a la derecha de los *plugins* indicando si se posee algún problema de compatibilidad al activar determinada configuración.

En el recuadro de la derecha se encuentra la información relativa al *plugin* seleccionado en el listado así cómo los botones de configuración específica del complemento y la ayuda relativa al mismo.

7.4.2.2. *Tools:*

La pestaña de herramientas (figura 70) permite la instalación de nuevos complementos desde la interfaz gráfica así como darle la una configuración paramétrica que se considere al mismo.

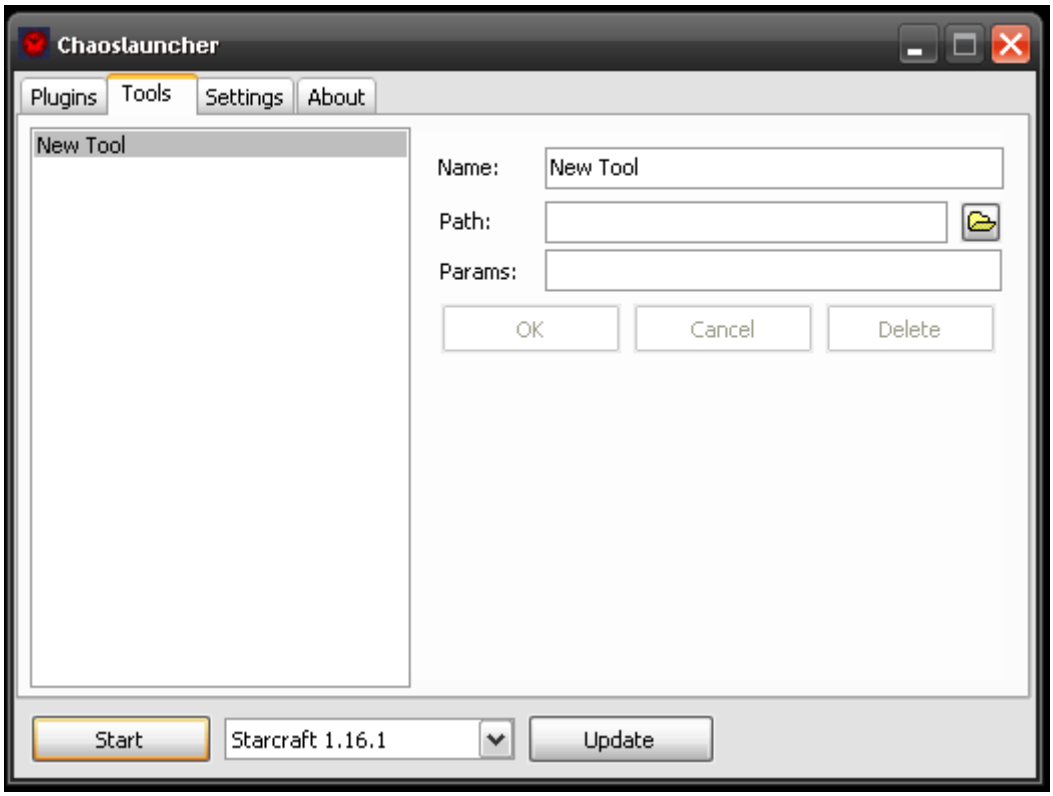

Figura 70: Herramientas de ChaosLauncher

En el recuadro de la izquierda se muestra el listado de herramientas instalado por los nombres que se haya especificado durante su instalación en el recuadro de la derecha. En este recuadro, además de asignarle un nombre, se especifica la ruta a la herramienta y los parámetros de ejecución del mismo.

7.4.2.3. *Settings:*

El panel de configuración (figura 71) permite modificar el comportamiento general de ChaosLauncher.

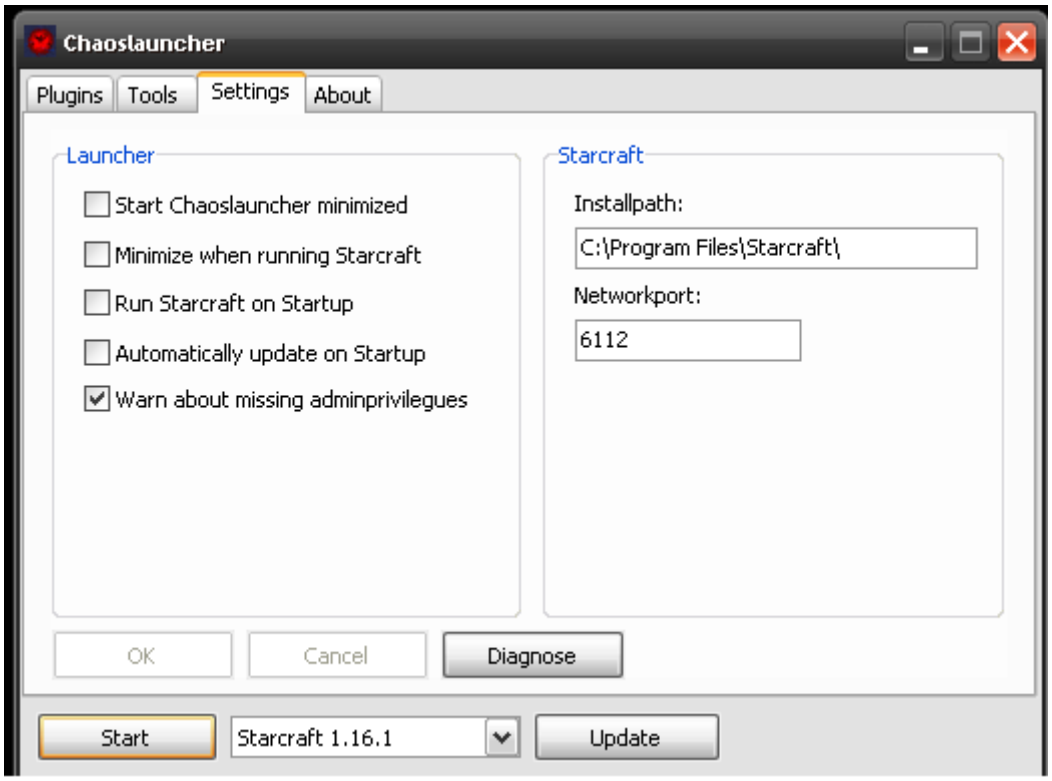

Figura 71: Configuración de ChaosLauncher

En el panel de la izquierda se muestran las opciones relativas a ChaosLauncher como casillas seleccionables. En el recuadro de la derecha se encuentra las opciones relativas a StarCraft, la ruta de instalación del juego y el puerto de comunicaciones que usará el juego. Finalmente, se dispone de un botón de diagnóstico para verificar que no existen problemas con la configuración del lanzador.

7.4.2.4. *About*

En el apartado 'acerca de' se muestra la información de la versión del lanzador y un enlace a las notas del mismo.

7.5. StarCraft

En este apartado se va a detallar una imagen global más precisa del juego describiendo las diferentes razas que lo componen e incidiendo particularmente en la Terran al mostrar su árbol tecnológico para edificios.

7.5.1. Razas

StarCraft presenta tres razas completamente diferentes entre sí que ofrecen un jugabilidad distinta en función de las características de las mismas.

7.5.1.1. *Protoss*

Los protoss son una raza alienígena antigua creada por los "*Xel'naga*"". Entre sus principales características destacan su avanzada tecnología, y su capacidad de comunicación vía psiónica. Se trata de una raza de unidades fuertes, caras y lentas, de gran poder individual y con una capacidad limitada de "regeneración", ya que disponen de escudos que los protegen y pueden ser recargados o recuperados. Es una raza eminentemente defensiva, con unidades limitadas en número y con alguna de las ellas capaz de hacerse invisible de forma permanente. Con la llegada de la expansión Broodwar, entre otras unidades se introdujo la posibilidad de fusión de dos unidades templarios oscuros para formar un arconte. Esta unidad es capaz de dominar la mente de una unidad enemiga y a partir de ese momento esa unidad entra a formar parte del ejército del jugador protoss con la característica de no gastar recursos de población del propio jugador, si se trata de otra raza, o superar el límite máximo de esta si es la propia.

7.5.1.2. *Terran*

Los Terran representan a la humanidad en el juego. Se trata de colonos que se perdieron en su misión desde la Tierra. Sus principales características son su versatilidad y la posibilidad de realizar complicadas maniobras de inteligencia. Como raza, representan un punto intermedio entre los ofensivos Zerg y los defensivos Protoss. Sus unidades no son tan lentas ni caras como los Protoss, pero resisten bastante menos. No disponen de escudos que puedan regenerarse, pero todas las unidades mecánicas pueden ser reparadas por un SCV, la unidad trabajador de los terran, con un coste bajo basado en el coste de la estructura a reparar. Con la llegada de la expansión Broodwar, se introdujo la unidad médica otorgando a los Terran la capacidad de regenerar unidades orgánicas aliadas incluso mientras estas pelean. Entre sus estrategias se encuentra el uso de herramientas de destrucción poderosas como el ataque nuclear, capaz de ser dirigido por una unidad especial de los Terran, el fantasma.

7.5.1.3. *Zerg*

Los Zerg son una raza insectoide alienígena creada por los "*Xel'naga*"' tras el fracaso de los Protoss. Poseen una mente de colmena y buscan la perfección genética a través la mutación y la adquisición de nuevos genotipos de otras especies. Es una raza rápida, versátil y muy numerosa en cantidad de unidades, si bien estas no son tan poderosas individualmente por norma general. Además, no disponen del apoyo de la tecnología. Estos defectos los suplen con su gran movilidad, su ingente número y su ferocidad. Además, poseen una capacidad de regeneración innata, capaz incluso de regenerar sus edificios de forma automática.

7.5.2. Árbol tecnológico: Terran

Este proyecto se ha centrado en la raza Terran y cómo en el resto de razas, los Terran presentan una serie de requisitos propios para acceder a edificios y unidades más avanzados así como para ciertas habilidades. En el caso de las habilidades que no posea la unidad de manera innata, se investigan en el edificio que permite el entrenamiento de dicha unidad tras un gasto en recursos y un tiempo en investigación. El árbol tecnológico para los edificios básicos se muestra en figura 72.

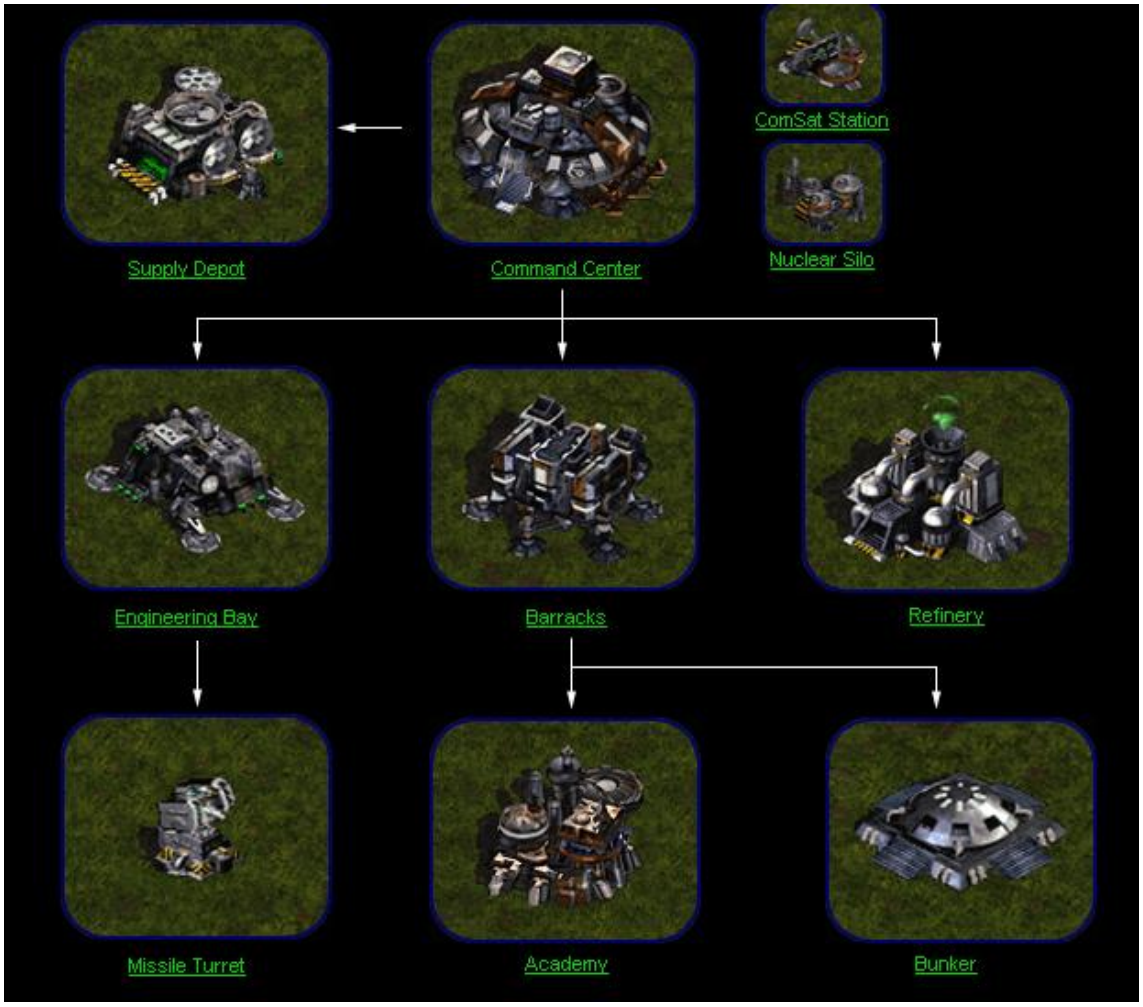

Figura 72: Árbol tecnológico básico Terran

Para los edificios avanzados, tras construir los cuarteles, el árbol tecnológico resultante se muestra en la figura 73.

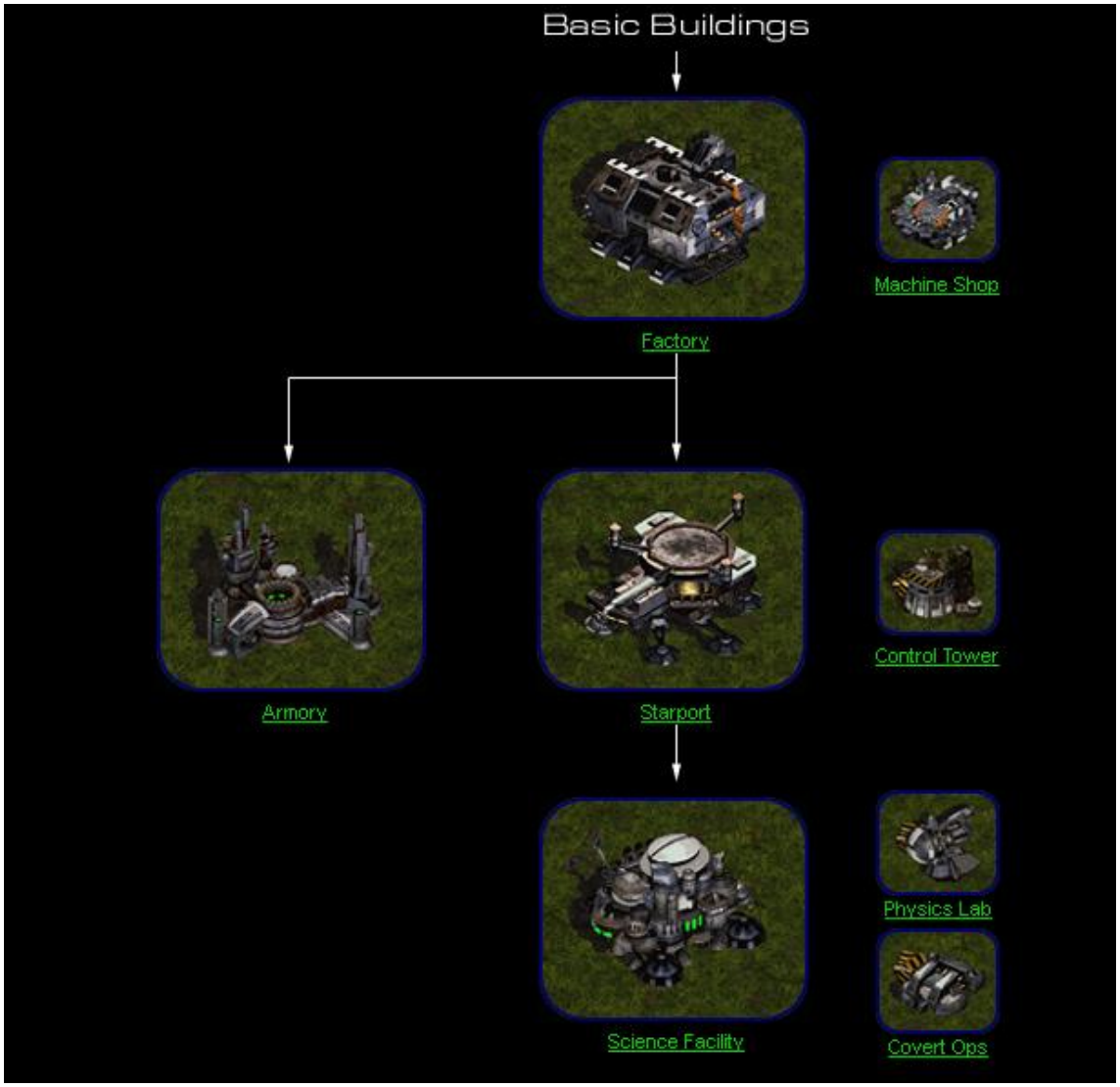

Figura 73: Árbol tecnológico avanzado Terran

Aquellos edificios no enlazados por flechas son edificios que se construyen como añadidos por el edificio principal al cuál quedan conectados y sin cuya conexión no pueden realizar su función. Sólo un edificio puede estar añadido al principal en cada momento.

Los siguientes edificios pueden despegar, adquirir estatus de unidad aérea, desplazarse y posteriormente establecerse en otra posición: Centro de mando, cuarteles, bahía de ingeniería, fábrica, puerto espacial y laboratorio de ciencia.

Las unidades que cada edificio proporciona son las siguientes:

- Centro de mando (*command center):*

- Vehículo de construcción espacial (SVC en sus siglas en inglés): es la unidad encargada de la recolección, construcción y reparación de los edificios Terran.

- Cuarteles (*barracks):*

  - Marine: infantería básica de ataque a distancia.

  - Murciélago de fuego*:* infantería de asalto de ataque a media distancia con ataque con área de efecto efectivo contra unidades de tamaño pequeño. Requiere poseer al menos una academia construida.

  - Fantasma: unidad de contraespionaje y agente de inteligencia, el fantasma es una unidad que se desbloquea al construir el edificio de operaciones encubiertas (*cover ops).* Su utilidad se basa en el guiado de misiles nucleares, sabotaje temporal de unidades mecánicas y por su capacidad de hacerse invisible a voluntad del jugador con un coste de energía que recupera en el tiempo.

  - Médico: esta unidad se puede producir en cuanto el edificio academia ha sido construida y su finalidad es restaurar los puntos de impacto de las unidades orgánicas con un coste de energía. Además puede restablecer unidades de estados alterados y aplicar ceguera, reduciendo así el rango de visión de la unidad afectada al mínimo.

- Fábrica (*factory*):

  - Buitre: unidad de movimiento rápido efectivo contra unidades de pequeño tamaño y por su habilidad de enterrar minas araña que persiguen unidades enemigas y explotan en contacto con ellas.

  - Tanque de asedio de arclite: se trata de un vehículo blindado que puede cambiar entre modo normal y un modo asedio que elimina su posibilidad de movimiento a cambio de un mayor rango de ataque y daño de área de efecto. Esta unidad se desbloquea cuando la tienda de máquinas está añadida a la fábrica.

  - Goliat: unidad mecánica bípeda caracterizada por disponer de misiles antiaéreos y cañones ametralladora para unidades terrestres. Se dispone de esta unidad cuando se ha construido una armería.

- Puerto espacial (*starport)*

  - Espectro: caza antiaéreo con ataque láser para unidades terrestres caracterizado por su habilidad para hacerse invisible a voluntad.

  - Transporte aéreo*:* nave capaz de transportar en su interior un determinado número de unidades en función de su tamaño. Carece por completo de ataque y requiere de la torre de control adosada al puerto espacial.

  - Crucero de batalla clase behemot: nave de grandes dimensiones de ataque lento pero efectivo contra unidades tanto aéreas como terrestres. Posee la habilidad de arma *yamato*, que permite lanzar un poderoso disparo a gran distancia con coste de energía. Esta unidad se habilita al construir el laboratorio de física y la torre de control.

- Vehículo de exploración científica: nave capaz de detectar unidades invisibles y de escudar unidades aliadas así como irradiar enemigas. Carece por completo de ataque y se habilita tras construir el laboratorio de ciencia.

- Fragata de misiles valquiria: unidad antiaérea sin ataque para unidades terrestres especializada en el combate de grandes concentraciones de unidades aéreas. Requiere de la armería y la torre de control adosada al puerto espacial.

7.6. Acrónimos y definiciones

Este anexo contiene una lista de los diferentes acrónimos que se han usado en el documento y las definiciones.

7.7. Acrónimos

- GNU: **G**NU **N**o es **U**nix.
- LAN: ***L**ocal **A**rea **N**etwork*. Red de área local.
- API: ***A**pplication **P**rogramming **I**nterface*. Interfaz de programación de aplicación.
- BWAPI: ***B**rood**W**ar **API***.
- BWTA: ***B**rood**W**ar **T**errain **A**nalizer*. Analizador de terreno de BroodWar.
- BWSAL: ***BW**API **S**tandar **A**ddon **L**ibrary*. Librería estándar de complementos de BWAPI.
- IPC*: **I**nternational **P**lanning **C**ompetition*. Competición internacional de planificación.
- HTML: ***H**yper**T**ext **M**arkup **L**anguage*. Lenguaje de marcado hipertextual.
- IDE: ***I**ntegrated **D**evelopment **E**nviroment*. Entorno de desarrollo integrado.
- DLL: ***D**inamyc-**L**ink **L**ibrary*. Biblioteca de enlace dinámico.
- GPL: ***G**eneral **P**ublic **L**icense*. Licencia pública general.
- FPS: **F**rames **P**er **S**econd. Fotogramas por segundo.
- PELEA: ***P**lanning, **L**earning and **E**xecution **A**rchitecture*. Arquitectura de planificación aprendizaje y ejecución.
- PLG: ***P**lanning & **L**earning **G**roup*. Grupo de planificación y aprendizaje.
- XML: *e**X**tensible **M**arkup **L**anguage*. Lenguaje de etiquetado extensible.
- RTS: ***R**eal **T**ime **S**trategy*. Estrategia en tiempo real.
- RAM: ***R**andom **A**ccess **M**emory*. Memoria de acceso aleatorio.
- CPU: ***C**entral **P**rocessing **U**nit*. Unidad de procesamiento central.
- GPU: ***G**raphic **P**rocessing **U**nit*, unidad de procesamiento gráfico.
- HDD: ***H**ard **D**isk **D**rive*, disco duro.

- Mhz: **M**ega-**h**ert**z**io.

- PDDL: ***P**lanning **D**omain **D**efinition **L**anguage*. Lenguaje de definición de dominios de planificación.

- XPDDL: **X**ML **PDDL**.

- PFC: **P**royecto de **F**in de **C**arrera.

- IA: **I**nteligencia **A**rtificial

- AIIDE: ***A**rtificial **I**ntelligence for **I**nteractive **D**igital **E**ntertainment.* Inteligencia artificial para el entretenimiento interactivo digital.

- OO: **O**rientación a **O**bjetos.

- OCW: ***O**pen **C**ourse **W**are*. Oferta de cursos abierta.

7.8. Definiciones

Bot: Abreviatura de robot, es un programa informático que realiza funciones muy diversas, imitando el comportamiento de un humano. En el ambiente de los videojuegos, se conoce como *bot* a programas que son capaces de jugar por sí mismos el juego.

StarCraft: es un videojuego de estrategia en tiempo real de ciencia ficción militar desarrollado por Blizzard Entertaiment® ampliamente reconocido por revolucionar el género de videojuegos de estrategia en tiempo real al contar con unidades exclusivas de cada raza que requieren comportamientos y estrategias diferentes. (40)

Apache-Tomcat: Es una implementación de software libre de las tecnologías Java Servlet y Java Server. La tecnología de Java Servlet proporciona a los desarrolladores web un mecanismo simple, sencillo y consistente para ampliar la funcionalidad de un servidor web y para acceder a sistemas de negocio existentes (41).

Exploit: Del inglés *to exploit*, *explotar* o *aprovechar*. Es un software, o una secuencia de comandos que busca crear un error en una aplicación, a fin de provocar un comportamiento imprevisto en los programas informáticos, hardware, o componente electrónico (42).

Script: En informática es un programa usualmente simple que se compone de una serie de instrucciones automáticas que son ejecutadas en orden (43).

Java: La tecnología Java es una plataforma y un lenguaje de programación orientada a objetos, robusto, portable, de alto nivel dinámico e interpretado (44).

Blizzard Entertaiment: Es una de las principales compañías de desarrollo y publicación de software de entretenimiento. Tras fundar la marca Blizzard® en 1994, la compañía se convirtió rápidamente en uno de los fabricantes de videojuegos más famosos y respetados. Al concentrar su energía en la creación de experiencias lúdicas excepcionales, Blizzard® ha sabido mantener una reputación de calidad sin igual desde su fundación (45).

Open source o código abierto: El código abierto es un método de desarrollo de software que posee el poder de la revisión distribuida y transparencia de proceso. La promesa del código abierto es mayor calidad, mayor fiabilidad, más flexibilidad, menor coste y el fin de los cierres predatorios de los vendedores (46).

Minería de datos: Es un conjunto de técnicas que surgen como consecuencia de la disponibilidad de grandes cantidades de datos y su complejidad de análisis manual. Su objetivo es convertir los datos en conocimiento para tomar decisiones (47) (48).

Sistema experto: Son sistemas donde se analiza la actividad de un experto humano cuando resuelve problemas en un área muy concreta y se intenta emular o ayudarle con capacidad para adquirir incrementalmente experiencia y capacidad para conversar con los usuarios y explicarles sus líneas de razonamiento (49).

ELO: Es un método para calcular la habilidad relativa entre jugadores de juegos de dos jugadores como el ajedrez. Su nombre proviene de su creador, Arpad Elo, un profesor de física Húngaro-Americano (50).

Host: Máquina conectada a una red. Tiene un nombre que la identifica, el *hostname*. La máquina puede ser una computadora, un dispositivo de almacenamiento por red, una impresora, etc… (51).

Malware: *Malware* o *software* de actividades ilegales es una categoría de código malicioso que incluye virus, gusanos y caballos de Troya. El malware destructivo utiliza herramientas de comunicación conocidas para distribuir gusanos que se envían por correo electrónico y mensajes instantáneos, caballos de Troya que provienen de ciertos sitios Web y archivos infectados de virus que se descargan de conexiones P2P. El malware también buscará explotar en silencio las vulnerabilidades existentes en sistemas (52).

Battle.net: Es un multijugador online proporcionado por Blizzard Entertaiment® (53).

Paradigma: filosofía o conjunto de suposiciones y/o técnicas que caracterizan una aproximación a una clase de problemas (54).

Arquitectura de control: descripción de un sistema a partir de componentes básicos y de cómo éstos encajan entre sí para formar el conjunto (54).

C++: Es un lenguaje de programación compilado, de tipado fuerte e inseguro, capaz de ser compatible con su código predecesor, C, y capaz de soportar múltiples paradigmas entre otras características (55).

Metric-FF: Es un sistema de planificación independiente de dominio desarrollado por Joerg Hoffmann. El sistema está desarrollado en C y es una extensión del planificador FF para combinar variables numéricas en el estado. Más precisamente PDDL 2.1 nivel 2 (13) .

Java RMI: La interfaz de método remoto, RMI por sus siglas en inglés, permite que un objeto ejecutando en una máquina virtual de Java invoque métodos en un objeto ejecutado en otra máquina virtual de Java. RMO provee de comunicación remota entre programas escritos en el lenguaje de programación Java (56).

Plugin: Programa que puede anexarse a otro para aumentar sus funcionalidades (generalmente sin afectar otras funciones ni afectar la aplicación principal). No es parche ni una actualización, es un módulo o complemento aparte que se incluye opcionalmente en una aplicación (57).

Inyección de código: Un ataque por inyección de código se produce cuando se fuerza a una aplicación a ejecutar un código no fiable para obtener una respuesta del programa utilizando para ello vulnerabilidades en la aplicación objetivo (58).

Referencias

Referencias

1. Lara-Cabrera, Raúl; Cotta, Carlos; y Fernández-Leiva, Antonio J. (2013) *A review of computational intelligence in RTS games. 2013* IEEE Symposium on Foundations of Computational Intelligence, M. Ojeda et al. (eds.), pp. 114-121, IEEE, Singapur.

2. Chung, Michael; Buro, Michael; y Schaeffer, Jonathan (2005). *Monte Carlo Planning in RTS Games.* Department of Computing Science, University of Alberta : s.n.

3. Rıza ERTÜRK, Umut (2009). Short Term Decision Making with Fuzzy Logic And Long Term Decision Making with Neural Networks In Real-Time Strategy Games. *Umut's weblog.* [En línea] 24 de 6 de 2009. [Citado el: 11 de 6 de 2013.] http://www.hevi.info/tag/artificial-intelligence-in-real-time-strategy-games/.

4. StarCraft. *Blizzard Entertaiment.* [En línea] [Citado el: 20 de 1 de 2013.] http://eu.blizzard.com/es-es/games/sc/.

5. M. Turing, Alan (1950). *COMPUTING MACHINERY AND INTELLIGENCE.* Mind 49: 433-460.

6. Smith, Chris; McGuire, Brian; Huang, Ting y Yang, Gary (2006) *The History of Artificial Intelligence.* University of Washington : s.n.

7. Borrajo Millán, Daniel (2008-2009). Automated Planning. *Open Course Ware .* [En línea]. [Citado el: 10 de 6 de 2013.] http://ocw.uc3m.es/ingenieria-informatica/automated-planning/slides/introduction.pdf/view.

8. Alcázar Saiz, Vidal; Guzmán, César; Prior, David; Borrajo, Daniel; Castillo, Luis; y Onaindía, Eva (2012). *pelea: Planning, Learning and Execution Architecture.* Universitat Politécnica de Valencia, Camino de Vera, s/n, 46022 Valencia, España; Universidad Carlos III de Madrid, Av. de la Universidad, 30, 28911 Leganés, España; Universidad de Granada, Av. Hospicio, s/n, 18010 Granada, España.

9. Weber, Ben, Mateas, Michael y Jhala, Arnav (2011). *Building Human-Level AI for Real-Time Strategy Games.* Expressive Intelligence Studio. UC Santa Cruz.

10. Arnold, Rachael (2007)**.** *Real Time Strategy Games as Domain for AI Research*. [En línea] [Citado el: 15 de 8 de 2013.] http://www.rachaelarnold.com/legacy/AI/. Hamilton College. CPSCI 370.

11. McDermott, Drew, Et al.(1998) *PDDL -- The Planning Domain Definition Language.* New Haven : s.n..

12. Hoffman, J.; y Nebel, B.(2001) *The FF planning system: Fast plan generation through heuristic search.* Journal of Articial Intelligence Research : s.n.

13. Hoffmann, Jörg, *Welcome to the Homepage of Metric-FF.* [En línea] [Citado el: 10 de 9 de 2012.] http://fai.cs.uni-saarland.de/hoffmann/metric-ff.html.

14. BWAPI. [En línea] [Citado el: 26 de 4 de 2013.] https://code.google.com/p/bwapi/.

15. Liquid, Team. *ChaosLauncher.* TEAMLIQUID WIKI. [En línea] [Citado el: 7 de 12 de 2012.] http://wiki.teamliquid.net/starcraft/Chaoslauncher.

16. Canonical. *Ubuntu*. [En línea] Canonical. [Citado el: 7 de 10 de 2012.] http://www.ubuntu.com/.

17. KDE**.** *KDE*. [En línea] [Citado el: 20 de 1 de 2013.] http://www.kde.org/.

18. Márquez Colás, Javier. *P01 - Registro de sucesos (log)*. Youtube*.* [En línea] 2013. [Citado el: 1 de 6 de 2013.] http://youtu.be/X9jF5x34Xvc.

19. Márquez Colás, Javier. *P02 - Configuración y conexión*. Youtube*.* [En línea] 2013. [Citado el: 1 de 6 de 2013.] http://youtu.be/QqL32p40Mrc.

20. Márquez Colás, Javier. *P03 - Envío de mensajes servidor/ cliente.* Youtube*.* [En línea] 2013. [Citado el: 1 de 6 de 2013.] http://youtu.be/Xp5KBLrU_jg.

21. Márquez Colás, Javier. *P04 - Ejecución de comando público de cliente desde el servidor*. Youtube*.* [En línea] 2013. [Citado el: 1 de 6 de 2013.] http://youtu.be/-LCPMgCbw28.

22. Márquez Colás, Javier. *P05 - Envío de mensajes cliente/servidor.* Youtube*.* [En línea] 2013. [Citado el: 1 de 6 de 2013.] http://youtu.be/XKQMqIFp5WM.

23. Márquez Colás, Javier. *P06 - Ejecución de comando público de servidor.* Youtube. [En línea] 2013. [Citado el: 1 de 6 de 2013.] http://youtu.be/lEAMCfK4WRg.

24. Márquez Colás, Javier. *P07 - Ejecución local servidor.* Youtube*.* [En línea] 2013. [Citado el: 1 de 6 de 2013.] http://youtu.be/nJPZl6hGjjk.

25. Márquez Colás, Javier. *P08 - Ejecución local cliente.* Youtube*.* [En línea] 2013. [Citado el: 1 de 6 de 2013.] http://youtu.be/zoa2nrWUeKU.

26. Márquez Colás, Javier. *P09 - Prueba de pérdida de conexión servidor*. Youtube*.* [En línea] 2013. [Citado el: 1 de 6 de 2013.] http://youtu.be/A5NDPP2TKC8.

27. Márquez Colás, Javier. *P10 - Prueba de acciones individuales: recolección total mineral.* Youtube*.* [En línea] 2013. [Citado el: 1 de 6 de 2013.] http://youtu.be/KCOw8AGSHg8.

28. Márquez Colás, Javier. *P11 - 12 - 18 Estado y funciones.* Youtube*.* [En línea] 2013. [Citado el: 1 de 6 de 2013.] http://youtu.be/dugHvDURalk.

29. Márquez Colás, Javier. *P13 - 15 Extracción de vespeno.* Youtube*.* [En línea] 2013. [Citado el: 1 de 6 de 2013.] http://youtu.be/T3FUDwdYZH4.

30. Márquez Colás, Javier. *P14 - Construcción de un edificio.* Youtube*.* [En línea] 2013. [Citado el: 1 de 6 de 2013.] http://youtu.be/2xsp7K8Nt1E.

31. Márquez Colás, Javier. *P16 - Creación de trabajadores*. Youtube*.* [En línea] 2013. [Citado el: 1 de 6 de 2013.] http://youtu.be/6LokZxifyto.

32. Márquez Colás, Javier. *P17 - Entrenamiento de unidades.* Youtube*.* [En línea] 2013. [Citado el: 1 de 6 de 2013.] http://youtu.be/uK9US9oZzPQ.

33. Márquez Colás, Javier. *P19 – Retrollamadas.* Youtube*.* [En línea] 2013. [Citado el: 1 de 6 de 2013.] http://youtu.be/bCTQFLr1DI8.

34. Márquez Colás, Javier. *P20 - Planificación con PELEA.* Youtube*.* [En línea] 2013. [Citado el: 1 de 6 de 2013.] https://www.youtube.com/watch?v=efzINxjGDeA.

35. Márquez Colás, Javier. *P21- 22 Planificación sin PELEA*. Youtube. [En línea] 2013. [Citado el: 1 de 6 de 2013.] https://www.youtube.com/watch?v=E8plVHls-8c.

36. Oracle. *Virtualbox*. [En línea] [Citado el: 7 de 10 de 2013.] https://www.virtualbox.org/.

37. Oracle. *Netbeans*. [En línea] [Citado el: 7 de 10 de 2012.] http://netbeans.org/.

38. *OpenVNC*. [En línea] [Citado el: 24 de 10 de 2012.] http://www.openvnc.com-about.com/.

39. *Notepad++*. [En línea] [Citado el: 24 de 10 de 2012.] http://notepad-plus-plus.org.

40. Blizzard Entertaiment. *StarCraft*. [En línea] [Citado el: 20 de 1 de 2013.] http://eu.blizzard.com/es-es/games/sc/.

41. Apache. *Apache Tomcat.* [En línea] [Citado el: 9 de 10 de 2012.] http://tomcat.apache.org/.

42. Pacheco, Federico G. y Jara, Hector (2012). *Ethical Hacking 2.0.* s.l. : Fox Andina.

43. Oxford Dictionaries. *Script*. [En línea] Oxford Dictionaries. [Citado el: 29 de 11 de 2012.] http://oxforddictionaries.com/us/definition/american_english/script.

44. Oracle**.** *About the Java Technology.* Oracle. [En línea] [Citado el: 29 de 11 de 2012.] http://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html, consultado.

45. Blizzard Entertaiment. *About Blizzard Entertaiment*. Blizzard. [En línea] [Citado el: 29 de 11 de 2012.] http://eu.blizzard.com/es-es/company/about/.

46. Open source. *Open Source Initiative*. Opensource. [En línea] [Citado el: 29 de 11 de 2012.] http://opensource.org/.

47. Aler, Ricardo. *Introducción al Aprendizaje Automático y a la Minería de Datos con Weka.* OCW. [En línea] [Citado el: 29 de 11 de 2012.] http://ocw.uc3m.es/ingenieria-informatica/herramientas-de-la-inteligencia-artificial/programa.

48. OCW. *Introducción al Aprendizaje Automático y a la Minería de Datos con Weka,* Herramientas de la inteligencia artificial Ingeniería informática. *OCW.* [En línea] [Citado el: 1 de 12 de 2012.] http://ocw.uc3m.es/ingenieria-informatica/herramientas-de-la-inteligencia-artificial/contenidos/transparencias/MDWEBHIA-clase.pdf.

49. Villena Román, Julio, Crespo García, Raquel M. y García Rueda, José Jesús**.** *Sistemas basados en conocimiento.* OCW. [En línea] [Citado el: 1 de 12 de 2012.] http://ocw.uc3m.es/ingenieria-telematica/inteligencia-en-redes-de-comunicaciones/material-de-clase-1/03-sistemas-basados-en-conocimiento.

50. Princenton. *Elo rating system*. [En línea] [Citado el: 15 de 12 de 2012.] http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Elo_rating_system.

51. Alegsa**.** *Host*. [En línea] [Citado el: 8 de 5 de 2013.] http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Elo_rating_system.

52. Norton**.** *Malware*. [En línea] [Citado el: 8 de 5 de 2013.] http://es.norton.com/security_response/malware.jsp.

53. Blizzard Entertaiment. *Terran strategy*. Battle.net. [En línea] 7 de 12 de 2012. http://classic.battle.net/scc/terran/bbuild.shtml.

54. OCW. *Planificación automática* . [En línea] 8 de 10 de 2012. [Citado el: 8 de 10 de 2012.] http://ocw.uc3m.es/ingenieria-informatica/planificacion-automatica/.

55. Cplusplus**.** *C++*. Cplusplus. [En línea] 28 de 4 de 2013. http://www.cplusplus.com/.

56. Oracle**.** *Trail RMI*. docs.oracle. [En línea] [Citado el: 8 de 5 de 2013.] http://docs.oracle.com/javase/tutorial/rmi/.

57. Alegsa**.** *Plugin*. [En línea] [Citado el: 5 de 8 de 2013.] http://www.alegsa.com.ar/Dic/plugin.php.

58. Donald, Ray y Ligatti (2012), Jay. Defining *Code-injection Attacks*, Department of Computer Science and Engineering. [En línea] [Citado el: 2 de 12 de 2012.] www.cse.usf.edu/~ligatti/papers/code-inj.pdf.